Pritam Suwal Shrestha (23771397)

Step 1: Understanding the Data

Datasets Overview

- **Economic Data**: Contains economic indicators like GDP, poverty headcounts, etc. Columns include Time, Country Name, Country Code, and various economic indicators like GDP, infant mortality rate, and internet security.
- **Global Population**: Population statistics by country or region. Columns span multiple years showing population data for various countries.
- **Life Expectancy**: Information on life expectancy per country or region. Data includes life expectancy rates per country across several years.
- **Countries by Continent**: Mapping countries to their respective continents. A simple mapping of countries to their respective continents.
- **Mental Illness**: Data regarding mental health statistics by country or region. Statistics related to mental health issues per country across different years.
- **Olympic Hosts:** Information on which countries hosted the Olympics and when. Information about Olympic games, including location, name, season, and year.
- **Olympic Medals:** Data on Olympic medals won by country. Detailed data on Olympic medals, including discipline, event, medal type, participant details, and country information.

Clients and their business queries

Based on the data available, I choose two clients that could query the available data for different insights.

1. Client A: National Olympic Committee (NOC) of the USA

- **Interest**: Investigating the correlation between the USA's economic factors and its Olympic medal haul, and analysing the influence of hosting the Olympics on these factors.

- **Analysis Approach**: This client would benefit from an analysis that correlates the 'Economic data.csv' and 'olympic_medals.csv' to explore how economic prosperity impacts Olympic success. Additionally, by integrating the 'olympic_hosts.csv', we can examine whether hosting the Games has any significant economic or performance-related impacts for the USA.

- **Importance**: This insight is crucial for understanding if and how economic strength and hosting the Games contribute to the USA's Olympic performance. It may influence future decisions on bids for hosting and investment in sports infrastructure and athlete development programs.

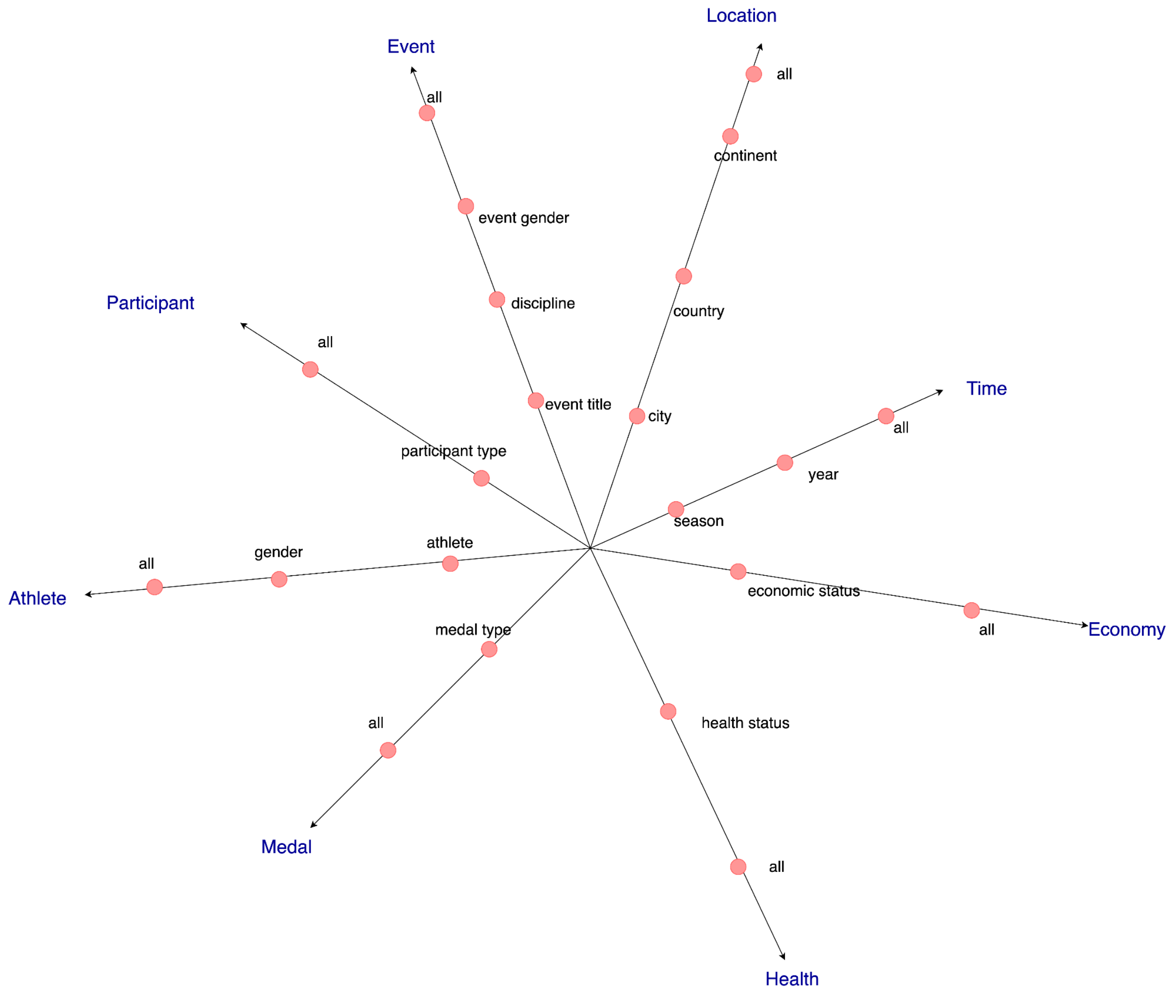
2. Client B: Australian Government Department of Health and Aged Care

- **Interest**:: Exploring the relationship between Australia's health metrics and its Olympic performance, and the impact of hosting the Olympics on these aspects.

- **Analysis Approach**: For this client, linking the 'mental-illness.csv' and 'life-expectancy.csv' with 'olympic_medals.csv' will reveal any correlations between national health and Olympic success. Incorporating data from 'olympic_hosts.csv' can also help analyse if hosting the Olympics has influenced these health metrics or the country's performance in the Games.

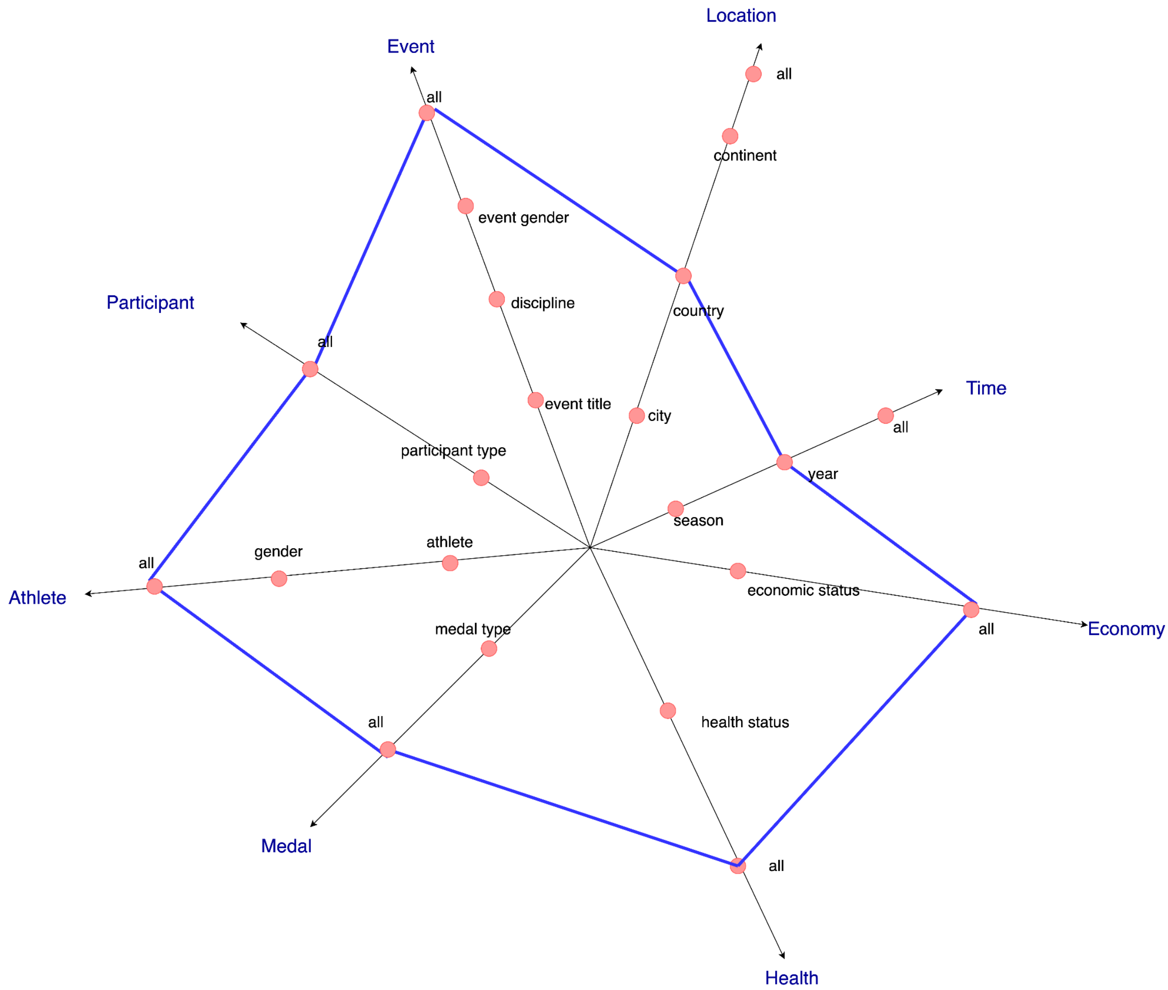
- **Importance**: This analysis is essential for understanding the interplay between national health and Olympic success. It can guide the formulation of public health policies and athlete support initiatives, especially in the context of preparing for or following up on hosting the Olympic Games.

Based on the above clients and the data available, I created a starnet model as follows. While hierarchy for some dimension looked obvious, Economy, Health and Medal dimension largely contains measures which are represented by economic status, health status, etc



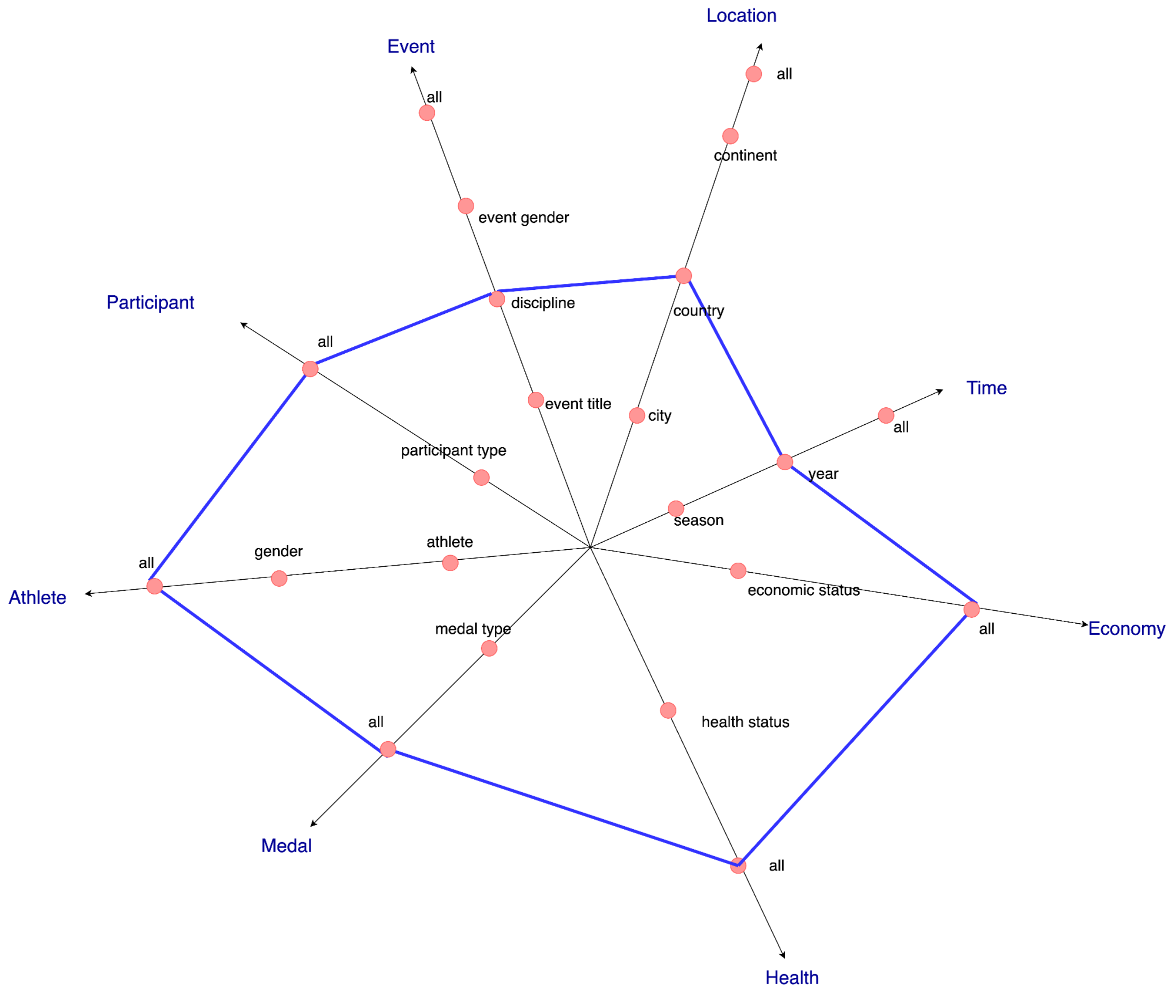
A. Potential Business Queries by National Olympic Committee (NOC) of the USA

1. **Medals vs. Hosting Years**: How did the USA's medal tally vary in Olympic Games immediately before, during, and after the years they hosted the Olympics?



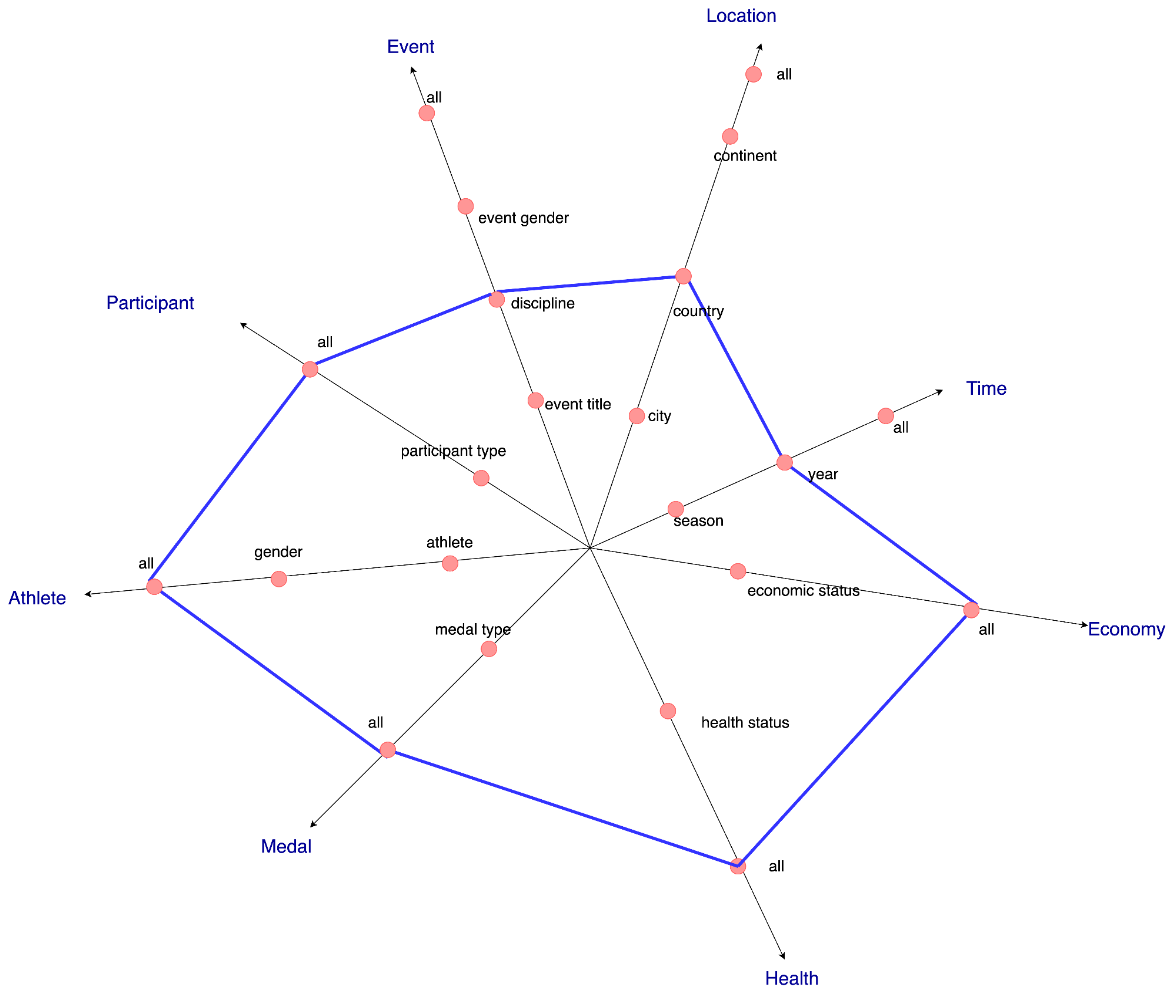
This question can be directly addressed with the Olympic medals dataset and the Olympic hosts dataset. We can track medal performance over the years and specifically during the hosting periods.

2. **Sport Discipline Success in Host Years**: Which sports disciplines yielded the most medals for the USA in the years they hosted the Olympics?



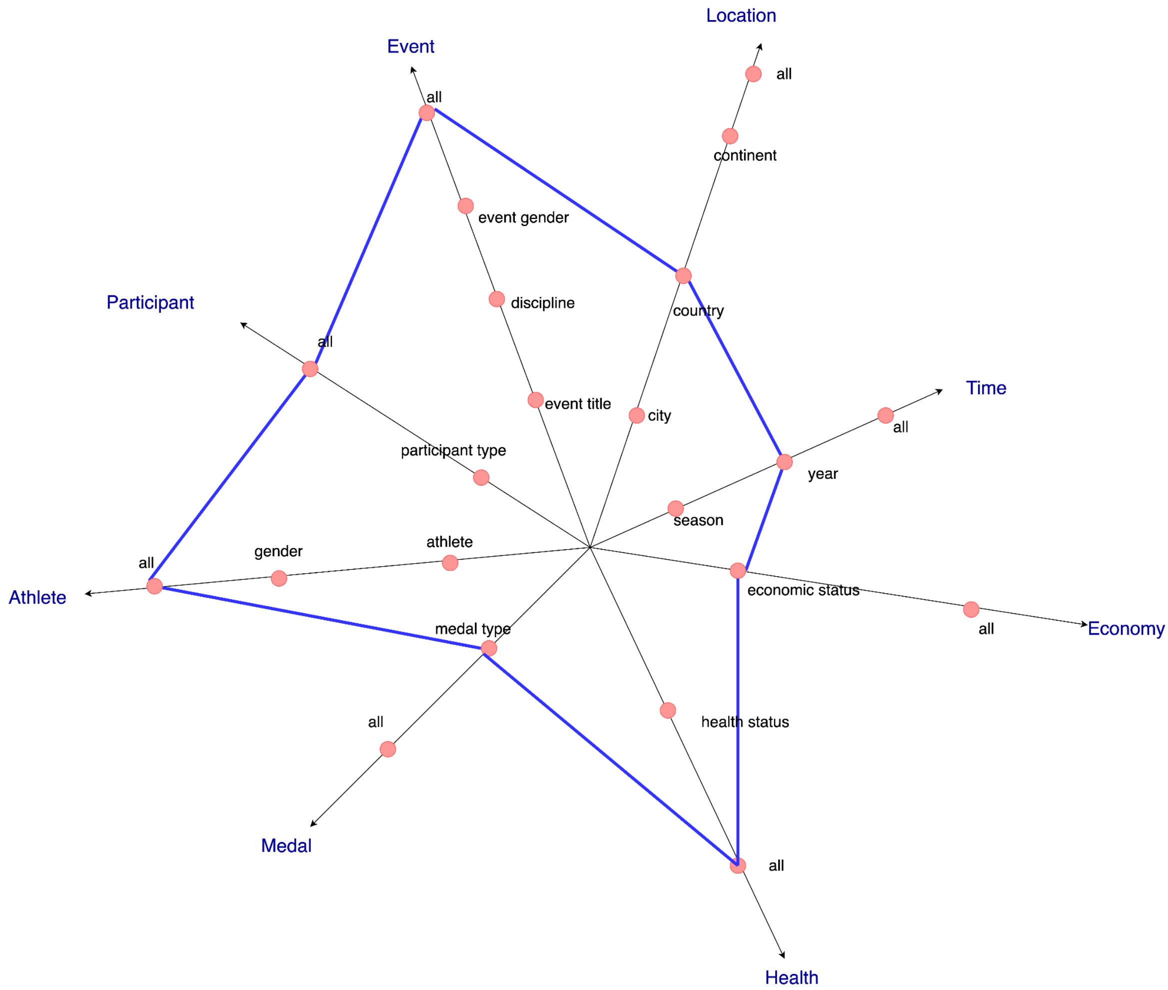
This question leverages the detail available in the Olympic medals dataset about specific sports disciplines and correlates it with hosting years from the Olympic hosts dataset.

3. **Economic Impact of Hosting**: What economic changes occurred in the USA in the years following their hosting of the Olympics (e.g., Los Angeles 1984, Atlanta 1996)?



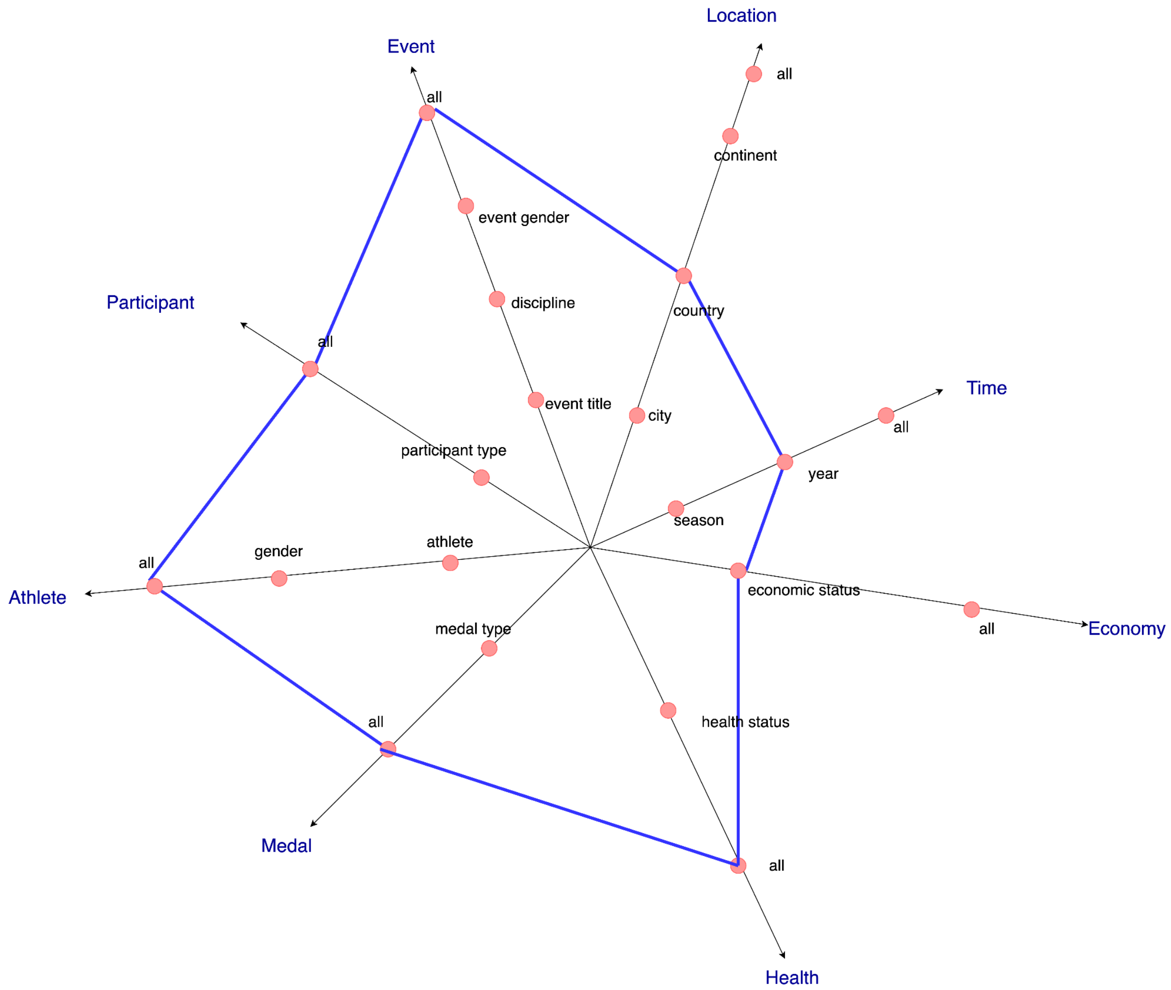
Provided the economic dataset includes data for the relevant years around each Olympic event hosted by the USA, this question can analyse economic trends and potential impacts resulting from hosting the Games. Since the economic data is only for 2020, we can get external data sources to answer this question. If not, we can only answer questions related to 2020.

4. **Economic Status and Medal Count:** Analyse the correlation between the USA's economic indicators in Olympic host years and their medal count in those years.



Using both the economic data and the Olympic medals dataset, this analysis can link economic conditions (like GDP per capita) to sports performance, particularly in the years the USA hosted the Olympics. We need external data sources in this case as well.

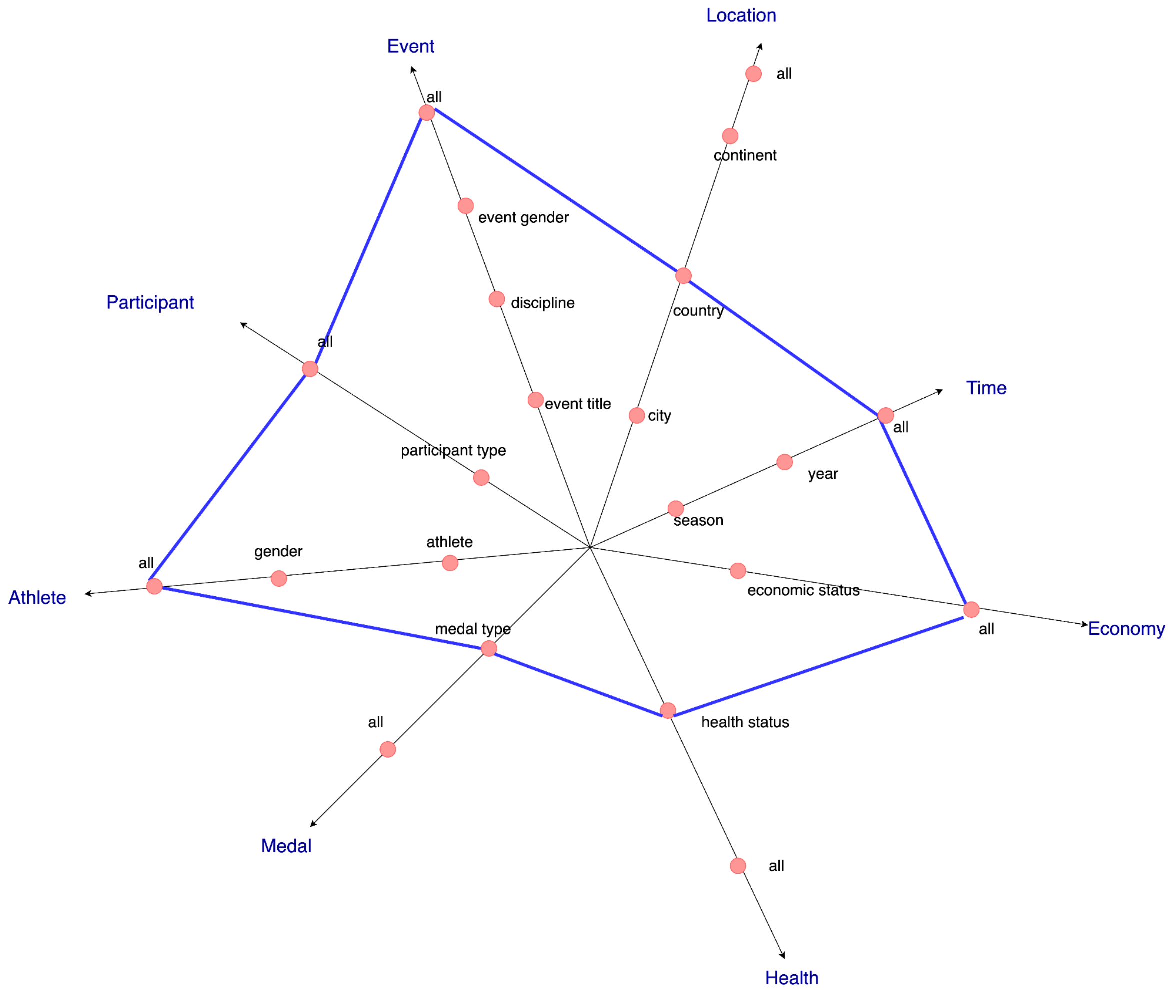
5. **Impact of Technological Advancement**: Does an increase in secure Internet servers per million people (as a proxy for technological advancement) in the USA correlate with a change in the total number of medals won?



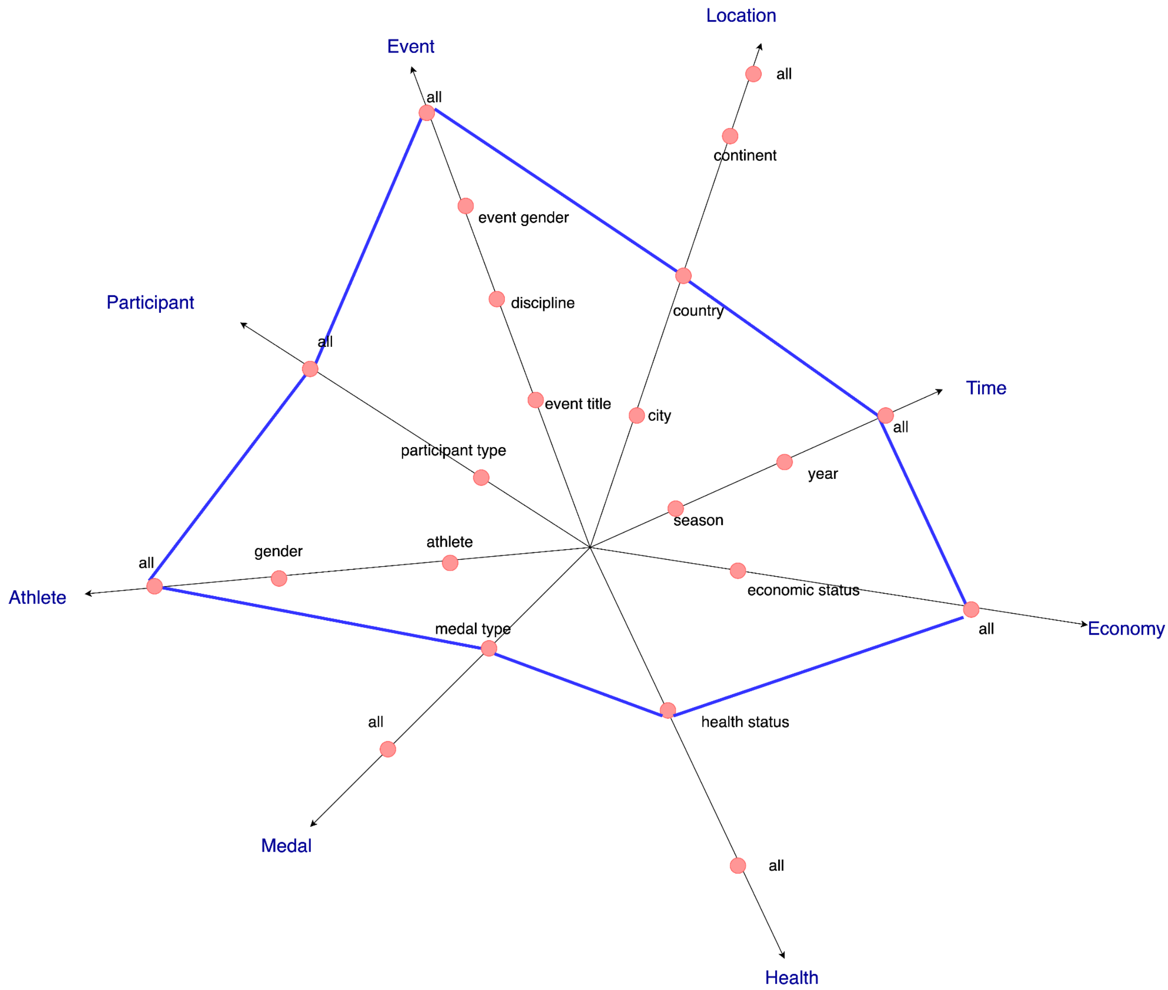
This question assumes the availability of data on technological advancements (like the number of secure internet servers) alongside Olympic performance data. If such data is tracked over similar timelines, it can provide insights into whether technological infrastructure improvements correlate with sports success. Additional data is needed to answer this question.

Potential Questions by Australian Government Department of Health and Aged Care

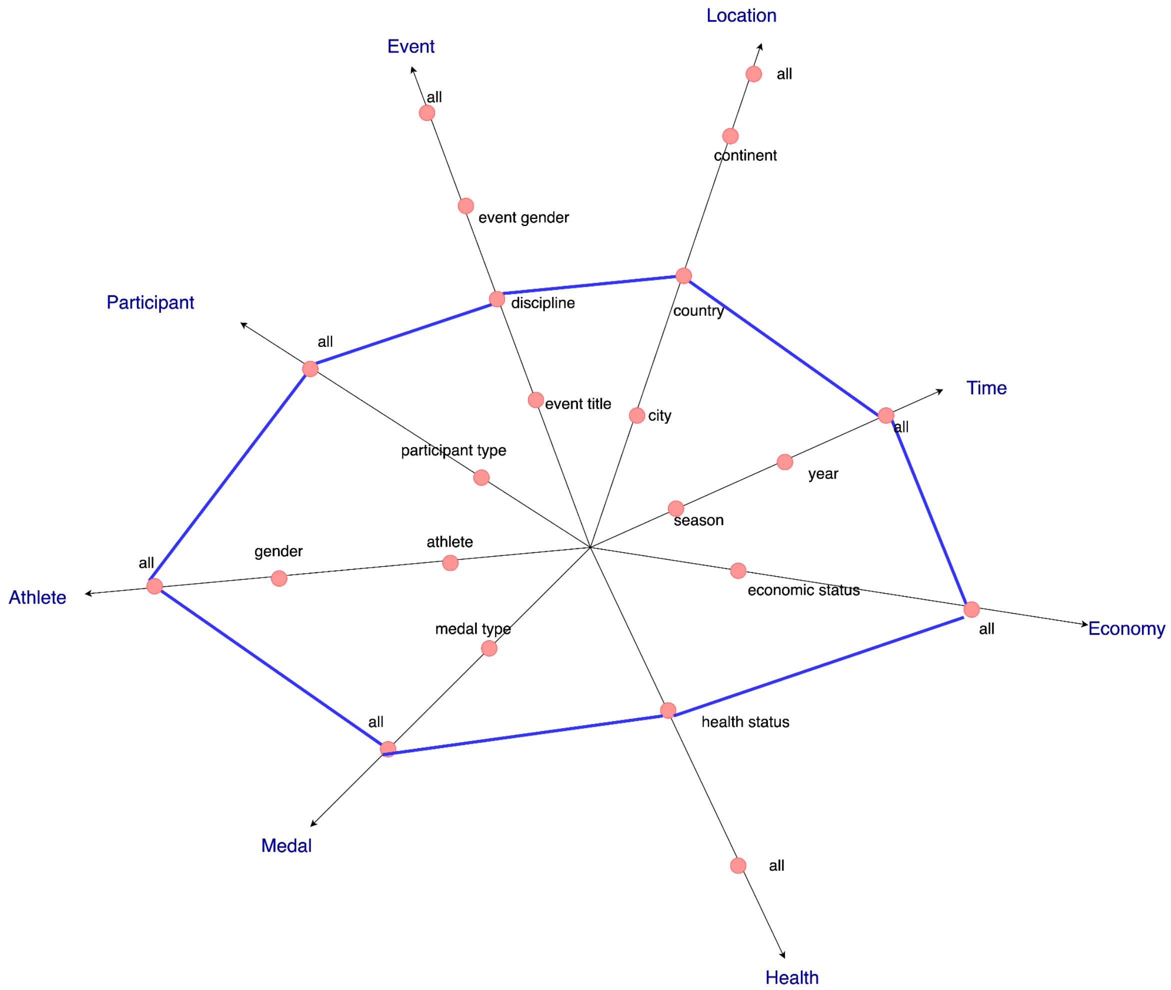
1. **Health and Performance Correlation**: Is there a relationship between Australia's overall health metrics (life expectancy, mental health status) and its Olympic medal tally?



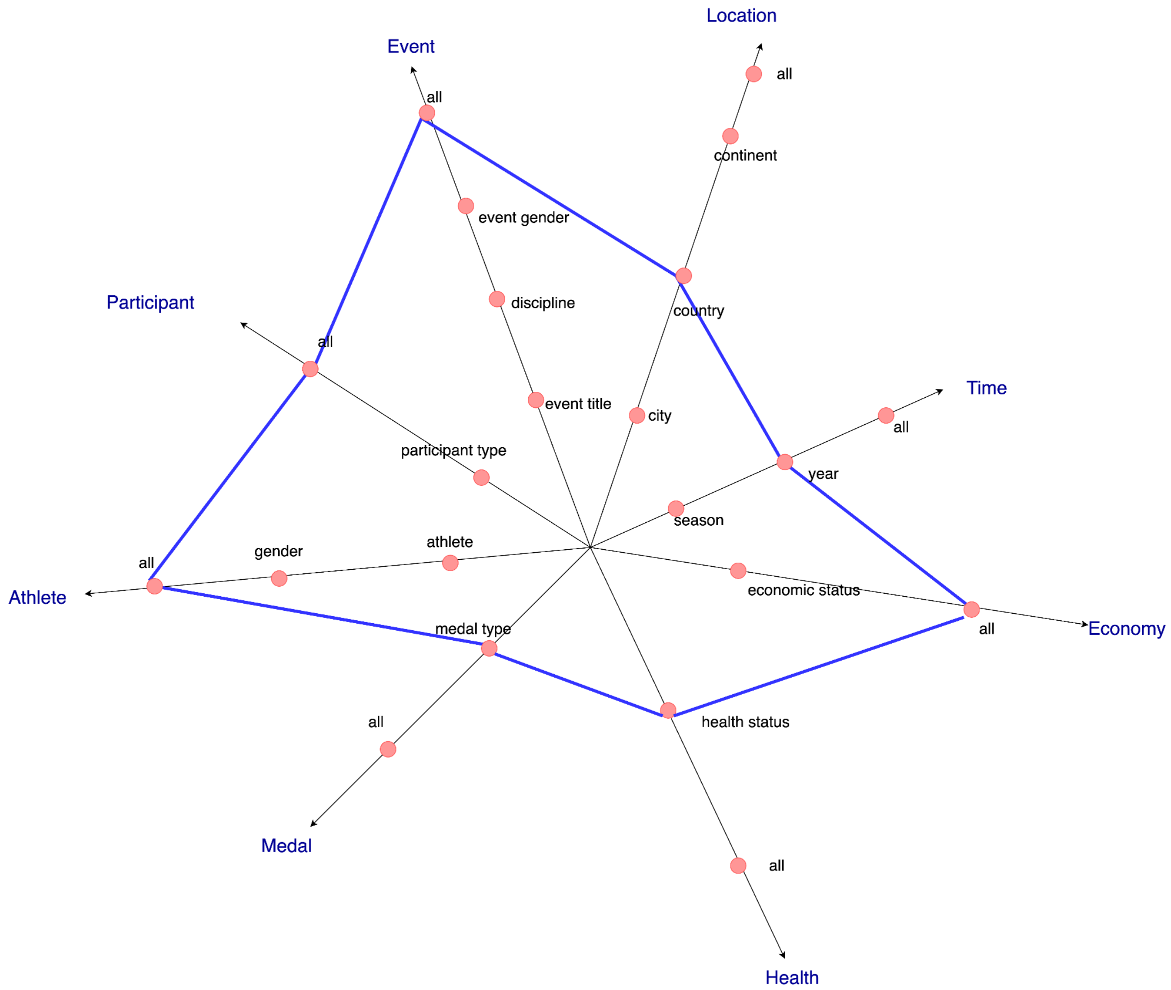
2. **Hosting Influence on Health and Performance**: Did hosting the Olympic Games (e.g., Sydney 2000) have any observable impact on national health metrics and subsequent Olympic performance?



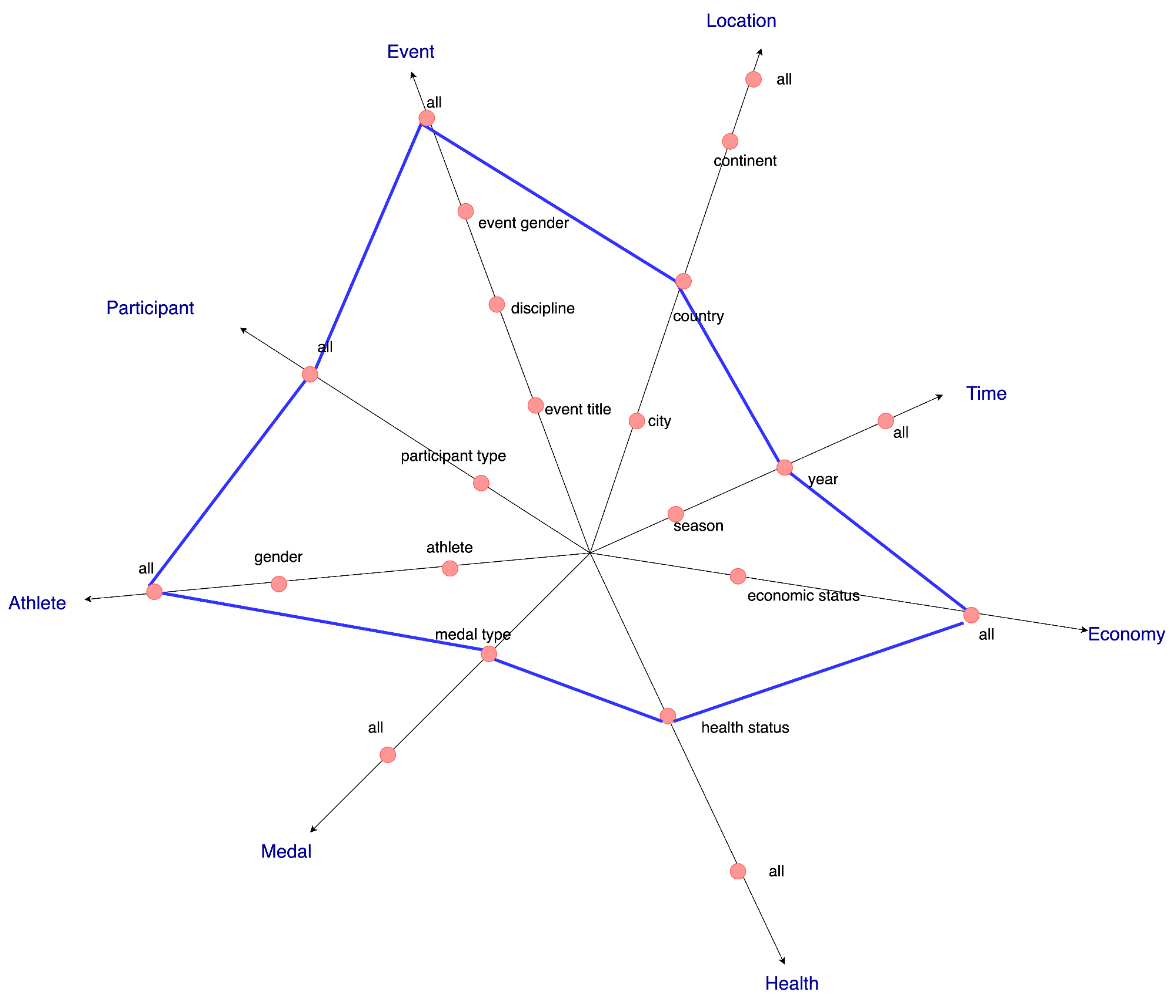
3. **Discipline-Specific Health Analysis**: Are certain sports disciplines more positively correlated with the national health status in terms of medal success?



4. **Long-Term Health Trends vs. Olympic Success**: How do long-term trends in health metrics in Australia correlate with long-term trends in Olympic success?



Comparative Health Analysis: How does Australia's Olympic success compare with that of other countries with similar health metrics?



Data Warehousing Design and Implementation

Following the four steps below of dimensional modelling (i.e. Kimball's four steps), design a data warehouse for the dataset(s).

1. **Identify the process being modelled.**

   The process here is the performance and participation of countries in the Olympic Games, taking into consideration various economic and health factors.

2. **Determine the grain at which facts can be stored.**

   The grain is the most detailed level at which facts are stored. In this case, it could be each individual Olympic event for which a medal was awarded. This would include the specific games, year, event, discipline, and the participating athletes/countries.

3. **Choose the dimensions**

   Based on the data above and the starnet model, I choose to create the following dimension table.

- DimYear
- DimEvent
- DimParticipant
- DimGame
- DimLocation
- DimAthlete

4. **Identify the numeric measures for the facts.**

   Possible measures could include:

   a. Number of medals (gold, silver, bronze)
   b. Economic indicators (GDP per capita, poverty rate)
   c. Health metrics (DALYs, life expectancy)

   So, I created the following fact tables which can contain these data.

   - FactOlympicMedalsMeasures
   - FactEconomicMeasure
   - FactHealthMeasure

Step 2: Data Profiling and Cleaning

Based on the above question and granularity we needed, here are some common cleaning steps we might consider for each dataset:

- **Handling Missing Values**: Determine how to handle rows or columns with a lot of missing data—whether to fill them, remove them, or keep them as is.
- **Standardising Formats**: Ensure that all data, especially categorical and date data, follow consistent formats across datasets.
- **Resolving Inconsistencies**: Look for and correct any discrepancies in naming conventions or data types, especially for fields that will serve as keys in our warehouse, like country names or codes.
- **Data Type Conversions:** Convert columns to the most appropriate data types (e.g., converting text to numeric where applicable).

Before doing anything, I have cleaned the data so that I don't have to worry much about data cleaning during the ETL process.

The detailed procedure for each data set is in the notebook. Following is the example for economic data.

Economic Data Cleaning Analysis

Key points from the initial analysis of the Economic Data:

- **Missing Data**: Several columns have a few missing values replaced by .. including country codes, economic indicators, and health expenditure data.
- **Data Types**: All columns are currently treated as object types (usually strings), which is not appropriate for numerical analysis. Columns representing monetary values, percentages, and ratios should be converted to numeric types.

Cleaning Steps for Economic Data:

1. **Handle Missing Values**:
    - Since the missing values are few, we can choose to fill these with appropriate placeholders such as the mean or median for continuous data, or we might choose to drop them if they pertain to critical fields like country codes where imputation is not advisable.
2. **Convert Data Types**:
    - Convert economic indicators from strings to floats to enable numerical operations.
    - Check for entries labelled as "no data" or ".." and treat them as NaN for appropriate numerical handling.

I'll start by replacing placeholders like ".." with NaN, converting data types, and handling missing values. I have run a similar script for all the dataset to profile the data. Which gives me following output to see the number of missing data which can be utilised in large dataset.

```
# Analyzing missing values and data types in the Economic Data dataset
economic_data = pd.read_csv('./raw_data/Economic data.csv')
economic_data_info = economic_data.info()
economic_data_missing_values = economic_data.isnull().sum()

economic_data_info, economic_data_missing_values
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 13 columns):
 #   Column                                                                       Non-Null Count  Dtype
---  ------                                                                       --------------  -----
 0   Time                                                                         202 non-null    object
 1   Time Code                                                                    200 non-null    object
 2   Country Name                                                                 200 non-null    object
 3   Country Code                                                                 200 non-null    object
 4   Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]  200 non-null    object
 5   GDP per capita (current US$) [NY.GDP.PCAP.CD]                                200 non-null    object
 6   GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]                         200 non-null    object
 7   Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]              200 non-null    object
 8   Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]              200 non-null    object
 9   Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]                    200 non-null    object
 10  Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED.PC.CD]  200 non-null    object
 11  Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]  200 non-null    object
 12  External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]     200 non-null    object
dtypes: object(13)
memory usage: 20.9+ KB

[2]:
(None,
 Time                                                                          3
 Time Code                                                                     5
 Country Name                                                                  5
 Country Code                                                                  5
 Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]  5
 GDP per capita (current US$) [NY.GDP.PCAP.CD]                                 5
 GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]                          5
 Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]               5
```

```
Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]                     5
Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]                           5
Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED.PC.CD]   5
Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]    5
External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]           5
dtype: int64)
```

Then I have proceeded to clean the data using the above mentioned methods.

```
# Replace placeholders with NaN
economic_data.replace(['..'], np.nan, inplace=True)

# Convert appropriate columns to numeric types
numeric_columns = economic_data.columns[4:]  # Columns from index 4
onwards are numeric
economic_data[numeric_columns] =
economic_data[numeric_columns].apply(pd.to_numeric, errors='coerce')

# Recheck missing values after conversions and update the dataset info
economic_data_missing_updated = economic_data.isnull().sum()
economic_data_info_updated = economic_data.info()

economic_data_missing_updated, economic_data_info_updated
```

After converting ".." to NaN, we can see there is significant number of data that's missing which was not visible when there was ".."

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 13 columns):
 #   Column                                                                              Non-Null Count  Dtype
---  ------                                                                              --------------  -----
 0   Time                                                                                202 non-null    object
 1   Time Code                                                                           200 non-null    object
 2   Country Name                                                                        200 non-null    object
 3   Country Code                                                                        200 non-null    object
 4   Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]  53 non-null     float64
 5   GDP per capita (current US$) [NY.GDP.PCAP.CD]                                      194 non-null    float64
 6   GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]                                191 non-null    float64
 7   Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]                    198 non-null    float64
 8   Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]                    194 non-null    float64
 9   Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]                          185 non-null    float64
 10  Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED.PC.CD]  186 non-null    float64
 11  Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]   185 non-null    float64
 12  External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]          167 non-null    float64
dtypes: float64(9), object(4)
memory usage: 20.9+ KB

[3]:
(Time                                                                                3
 Time Code                                                                           5
 Country Name                                                                        5
 Country Code                                                                        5
 Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]  152
 GDP per capita (current US$) [NY.GDP.PCAP.CD]                                      11
 GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]                                14
 Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]                    7
 Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]                    11
 Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]                          20
 Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED.PC.CD]  19
 Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]   20
 External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]           38
 dtype: int64,
 None)
```

I then imputed the missing values with the median.

```
# Calculate median for numeric columns
median_values = economic_data[numeric_columns].median()


# Fill missing values
economic_data[numeric_columns] =
economic_data[numeric_columns].fillna(median_values)

# Drop rows where 'Country Name' or 'Country Code' is missing
economic_data.dropna(subset=['Country Name', 'Country Code'],
inplace=True)

final_missing_values_pandas = economic_data.isnull().sum()
final_data_preview_pandas = economic_data.head()

final_missing_values_pandas
```

After that there was no any empty data.

```
Time                                                                                      0
Time Code                                                                                 0
Country Name                                                                              0
Country Code                                                                              0
Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]         0
GDP per capita (current US$) [NY.GDP.PCAP.CD]                                             0
GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]                                      0
Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]                           0
Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]                           0
Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]                                 0
Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED.PC.CD]   0
Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]          0
External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]                  0
dtype: int64
```

Then I loaded the data to OLTP after creating the connection to Postgrse database**.**

```
from sqlalchemy import create_engine

connection_url = f"postgresql://{db_user}:{db_password}@{db_host}:{db_port}/{db_name}"

# Create the engine
engine = create_engine(connection_url)

create_economic_data_table_sql = """
CREATE TABLE IF NOT EXISTS economic_data (
    time_year VARCHAR(255),
    time_code VARCHAR(255),
    country_name VARCHAR(255),
    country_code VARCHAR(255),
    poverty_ratio FLOAT, -- Ratio of population at $2.15 a day PPP
    gdp_per_capita_usd FLOAT, -- GDP per capita in current US$
```

```
    gdp_per_capita_growth FLOAT, -- Annual growth of GDP per capita
    secure_internet_servers_per_million FLOAT, -- Secure Internet servers per million people
    infant_mortality_rate FLOAT, -- Infant mortality rate per 1,000 live births
    health_expenditure_pct_gdp FLOAT, -- Current health expenditure as % of GDP
    gov_health_expenditure_per_capita_usd FLOAT, -- Government health expenditure per capita in current US$
    private_health_expenditure_per_capita_usd FLOAT, -- Private health expenditure per capita in current US$
    external_health_expenditure_per_capita_usd FLOAT -- External health expenditure per capita in current US$
);
"""

cursor = connection.cursor()

cursor.execute(create_economic_data_table_sql)

connection.commit()
cursor.close()
connection.close()
```

I have used the same pattern to load the data to the database in the entire notebook.

```
economic_data.to_sql("economic_data", con=engine, if_exists="append",
index=False)
```

I also saved the cleaned data in the csv file if needed to access later one.

```
economic_data.to_csv('cleaned_data/Economic data.csv', index=False)
```

For a **Global Population** with a lot of missing data, **linear interpolation** was used to fill the missing data as the population usually has gradual linear changes.

```
# Interpolate missing data for each country
global_population_data.iloc[:, 1:] = global_population_data.iloc[:,
1:].apply(lambda x: x.interpolate(method='linear',
limit_direction='both'), axis=1)
```

ETL Process

First I created a [Logical Map](https://uniwa-my.sharepoint.com/:x:/g/personal/23771397_student_uwa_edu_au/Eb6NHS4zK49KrIK9nhU1PWoBAOnKYRNz4zx6AEI0braTWQ?e=nXA5oK) to assist myself during the ETL process and creation of Dimension and Fact tables. The link can be found here (**https://uniwa-my.sharepoint.com/:x:/g/personal/23771397_student_uwa_edu_au/Eb6NHS4zK49KrIK9nhU1PWoBAOnKYRNz4zx6AEI0braTWQ?e=nXA5oK**)

A	B	C	D	E	F	G	H	I
TARGET								
Target Table	**Target Column**	**Nullable**	**PK/FK**	**DataType**	**Remarks**	**Source CSV**	**Source Column**	**DataType**
DimLocation	country_code	NO	PK	CHAR(3)	- Two types of country codes: two characters and three characters. - Decision to use three-character codes due to: - No null values in the list.	olympic_medals.csv	country_3_letter_code	string
	country_name			VARCHAR	Name mismatch, some countries did not participate in olympic	olympic_medals.csv, olympic_hosts.csv, mental_illness_countries.csv, life_expectancy_countries.csv ecnomic_data.csv	Country Name	
DimEvent	event_id	no	PK	integer				
	event_title			VARCHAR		olympic_medals.csv	event_title	
	event_discipline			VARCHAR		olympic_medals.csv	discipline	
	event_gender			VARCHAR		olympic_medals.csv	enent_gender	
DimParticipant	participant_id		PK	integer				
	participant_title			VARCHAR		olympic_medals.csv	participant_title	
	participant_type			VARCHAR	Athlete/Team	olympic_medals.csv	participant_type	
DimAthlete	athlete_id		PK	INTEGER				
	athlete_name			VARCHAR		olympic_medals.csv	athlete_full_name	
	athlete_url			VARCHAR		olympic_medals.csv	athlete_url	
	athlete_gender			VARCHAR	men, women, open we can put null when the event _gender is open so it can be manually updated later	olympic_medals.csv	event_gender	
DimYear	year	NO	PK	INTEGER	unique years since start of olympic 1896 - 2028	olympic_medals.csv, Global Population.csv	year	
DimGame	game_slug	NO	Pk	VARCHAR		olympic_hosts.csv		
	game_name			VARCHAR		olympic_hosts.csv		
	game_season			VARCHAR		olympic_hosts.csv		
	game_year			INTEGER		olympic_hosts.csv		
	country_code			char				
FactOlympicMedalsMeasures	game_slug	NO	FK (DimGame)					
	participant_id		FK (DimParticipant)					
	athlete_id		FK (DimAthlete)					
	event_id	NO	FK (DimEvent)					
	country_code	NO	FK (DimLocation)					
	year	NO	FK (DimYear)	INTEGER				
	bronze_medals			INTEGER		olympic_medals.csv		
	silver_medals			INTEGER		olympic_medals.csv		
	gold_medals			INTEGER		olympic_medals.csv		
FactEconomicMeasure	year	NO	FK (DimYear)					
	country_code	NO	FK (DimLocation)	CHAR				
	poverty_count			FLOAT				
	gdp_per_capita			FLOAT				
	annual_gdp_growth			FLOAT				
	servers_count			INTEGER				
FactHealthMeasure	year	NO	FK	INTEGER				
	country_code	NO	FK	CHAR				
	daly_depression			FLOAT				
	daly_schizophrenia			FLOAT				
	daly_bipolar_disorder			FLOAT				
	daly_eating_disorder			FLOAT				
	daly_anxiety			FLOAT				
	life_expectancy			FLOAT				
	infant_mortality_rate			FLOAT				
	current_health_expenditure			FLOAT				
	government_health_expenditure			FLOAT				
	private_health_expenditure			FLOAT				
	external_health_expenditure			FLOAT				

Please refer to the attached notebook for the complete ETL process because it has more details.

Cleaning Countries Information.

This was the most extensive part of the project as there were different names and codes for the same country. I created an excel sheet to keep track of the countries and used external data sources to create standard names and codes.

Some countries have different old and new country code as follows.Manually created with comparison form the online resources

Old Code	New Code	Country Name	Remarks
AHO	NLD	Netherlands Antilles	Dissolved in 2010
ALG	DZA	Algeria	
ANZ	None	Australia and New Zealand	Historical context, no single current ISO code
BAH	BHS	Bahamas	
BAR	BRB	Barbados	
BER	BMU	Bermuda	
BOH	CZE	Bohemia	Historical region, now part of Czech Republic
BOT	BWA	Botswana	
BUL	BGR	Bulgaria	
BUR	MMR	Burma	Now Myanmar
CHI	CHL	Chile	
CRC	CRI	Costa Rica	
CRO	HRV	Croatia	
DEN	DNK	Denmark	
EUN	None	Unified Team	Represented former Soviet Union republics in 1992
FIJ	FJI	Fiji	
FRG	DEU	Federal Republic of Germany	Now Germany
GDR	DEU	German Democratic Republic	Now part of Germany
GER	DEU	Germany	
GRE	GRC	Greece	
GRN	GRD	Grenada	
GUA	GTM	Guatemala	
HAI	HTI	Haiti	
INA	IDN	Indonesia	
IOA	None	Independent Olympic Athletes	No standard ISO code
IRI	IRN	Iran	
ISV	VIR	Virgin Islands, U.S.	
KOS	XKX	Kosovo	Not universally recognized
KSA	SAU	Saudi Arabia	
KUW	KWT	Kuwait	
LAT	LVA	Latvia	
MAS	MYS	Malaysia	
MGL	MNG	Mongolia	
MIX	None	Mixed team	No standard ISO code
MRI	MUS	Mauritius	
NED	NLD	Netherlands	
NGR	NGA	Nigeria	
NIG	NER	Niger	
OAR	None	Olympic Athletes from Russia	No standard ISO code
PAR	PRY	Paraguay	
PHI	PHL	Philippines	
POR	PRT	Portugal	
PUR	PRI	Puerto Rico	
ROC	TWN	Taiwan, Republic of China	Commonly used ISO code is TWN for Taiwan

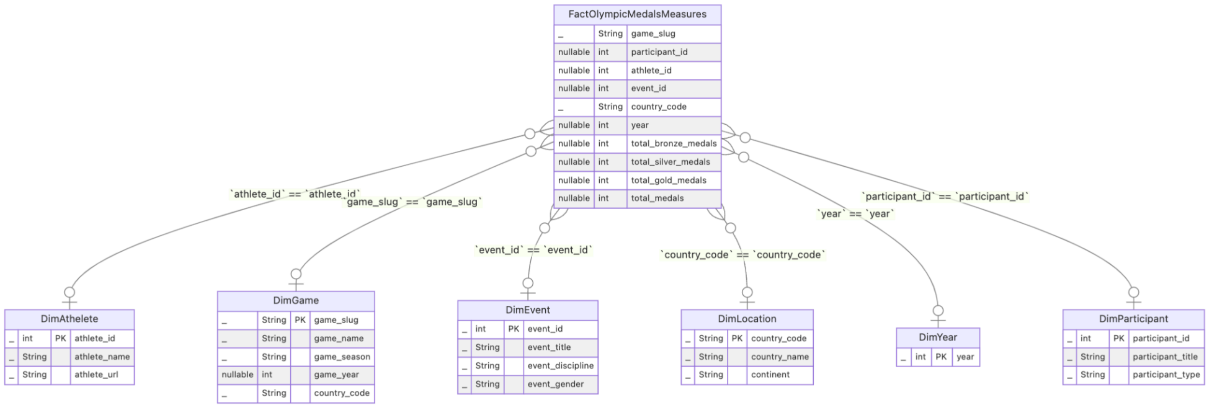
RSA	ZAF	South Africa	
SAM	WSM	Samoa	
SCG	SRB/MNE	Serbia and Montenegro	Dissolved, now Serbia SRB and Montenegro MNE
SLO	SVN	Slovenia	
SRI	LKA	Sri Lanka	
SUD	SDN	Sudan	
SUI	CHE	Switzerland	
TAN	TZA	Tanzania	
TCH	CZE/SVK	Czechoslovakia	Now Czech Republic CZE, and Slovakia SVK
TGA	TON	Tonga	
TOG	TGO	Togo	
TPE	TWN	Chinese Taipei	Commonly used ISO code is TWN for Taiwan
UAE	ARE	United Arab Emirates	
UAR	EGY	United Arab Republic	Dissolved, was a union between Egypt and Syria
URS	RUS	Soviet Union	Dissolved, the largest successor state is Russia
URU	URY	Uruguay	
VIE	VNM	Vietnam	
WIF	None	West Indies Federation	Dissolved, was a political union of Caribbean islands
YUG	SRB/HRV	Yugoslavia	Dissolved, successor states include Serbia SRB, Croatia HRV, etc.
ZAM	ZMB	Zambia	
ZIM	ZWE	Zimbabwe	

Step 3: Data Modelling

I created a schema for the fact table and dimension table we have using atoti. This code generated the schema shown in the diagram below.

```
olympic_medals_measures.join(dimlocation_table, olympic_medals_measures["country_code"] ==
dimlocation_table["country_code"])

olympic_medals_measures.join(dimevent_table, olympic_medals_measures["event_id"] ==
dimevent_table["event_id"])

olympic_medals_measures.join(dimparticipant_table, olympic_medals_measures["participant_id"] ==
dimparticipant_table["participant_id"])

olympic_medals_measures.join(dimathlete_table, olympic_medals_measures["athlete_id"] ==
dimathlete_table["athlete_id"])

olympic_medals_measures.join(dimyear_table, olympic_medals_measures["year"] ==
dimyear_table["year"])

olympic_medals_measures.join(dimgame_table, olympic_medals_measures["game_slug"] ==
dimgame_table["game_slug"])
```

7]:



References:

ChatGPT has been used to generate some codes in the process and get insights on how to tackle some debugging issues.

Step 1: Understanding the Data

Datasets Overview

1. **Economic Data**: Contains economic indicators like GDP, inflation rates, etc. Columns include Time, Country Name, Country Code, and various economic indicators like GDP, infant mortality rate, and internet security.
2. **Global Population**: Population statistics by country or region. Columns span multiple years showing population data for various countries.
3. **Life Expectancy**: Information on life expectancy per country or region. Data includes life expectancy rates per country across several years.
4. **Countries by Continent**: Mapping countries to their respective continents. A simple mapping of countries to their respective continents.
5. **Mental Illness**: Data regarding mental health statistics by country or region. Statistics related to mental health issues per country across different years.
6. **Olympic Hosts**: Information on which countries hosted the Olympics and when. Information about Olympic games, including location, name, season, and year.
7. **Olympic Medals**: Data on Olympic medals won by country. Detailed data on Olympic medals, including discipline, event, medal type, participant details, and country information.

Step 2: Data Profiling and Cleaning

I will load each dataset and perform basic profiling to identify issues like missing values, data types, and potential primary keys.

Here are some common cleaning steps we might consider for each dataset:

1. **Handling Missing Values**: Determine how to handle rows or columns with a lot of missing data—whether to fill them, remove them, or keep them as is.
2. **Standardizing Formats**: Ensure that all data, especially categorical and date data, follow consistent formats across datasets.
3. **Resolving Inconsistencies**: Look for and correct any discrepancies in naming conventions or data types, especially for fields that will serve as keys in our warehouse, like country names or codes.
4. **Data Type Conversions**: Convert columns to the most appropriate data types (e.g., converting text to numeric where applicable).

1. Economic Data

I'll start by analyzing the missing values and data types in the Economic Data dataset. After that, we can move on to the other datasets. Let's review the Economic Data first. The last few rows are deleted manually because they are just some meta information which we don't require.

In [1]:

```
import pandas as pd
import numpy as np
```

In [2]:

```
# Analyzing missing values and data types in the Economic Data dataset
economic_data = pd.read_csv('./raw_data/Economic data.csv')
economic_data_info = economic_data.info()
economic_data_missing_values = economic_data.isnull().sum()

economic_data_info, economic_data_missing_values
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
```

```
Data columns (total 13 columns):
 #   Column
Non-Null Count  Dtype
---  ------
--------------  -----
 0   Time
202 non-null    object
 1   Time Code
200 non-null    object
 2   Country Name
200 non-null    object
 3   Country Code
200 non-null    object
 4   Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]
200 non-null    object
 5   GDP per capita (current US$) [NY.GDP.PCAP.CD]
200 non-null    object
 6   GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]
200 non-null    object
 7   Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
200 non-null    object
 8   Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]
200 non-null    object
 9   Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]
200 non-null    object
 10  Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED
.PC.CD]  200 non-null    object
 11  Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]
200 non-null    object
 12  External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]
200 non-null    object
dtypes: object(13)
memory usage: 20.9+ KB
```

Out[2]:

```
(None,
 Time
3
 Time Code
5
 Country Name
5
 Country Code
5
 Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]
5
 GDP per capita (current US$) [NY.GDP.PCAP.CD]
5
 GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]
5
 Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
5
 Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]
5
 Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]
5
 Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED.PC.
CD]     5
 Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]
5
 External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]
5
 dtype: int64)
```

Economic Data Cleaning Analysis

Key points from the initial analysis of the Economic Data:

- **Missing Data: Several columns have a few missing values repalced by .. including country codes, economic indicators, and health expenditure data.**

- **Data Types: All columns are currently treated as object types (usually strings), which is not appropriate for numerical analysis. Columns representing monetary values, percentages, and ratios should be converted to numeric types.**

Cleaning Steps for Economic Data:

1. **Handle Missing Values:**
   - **Since the missing values are few, we can choose to fill these with appropriate placeholders such as the mean or median for continuous data, or we might choose to drop them if they pertain to critical fields like country codes where imputation is not advisable.**
2. **Convert Data Types:**
   - **Convert economic indicators from strings to floats to enable numerical operations.**
   - **Check for entries labeled as "no data" or ".." and treat them as `NaN` for appropriate numerical handling.**

I'll start by replacing placeholders like ".." with NaN, converting data types, and handling missing values.

In [3]:

```
# Replace placeholders with NaN
economic_data.replace(['..'], np.nan, inplace=True)

# Convert appropriate columns to numeric types
numeric_columns = economic_data.columns[4:]  # Columns from index 4 onwards are numeric
economic_data[numeric_columns] = economic_data[numeric_columns].apply(pd.to_numeric, erro
rs='coerce')

# Recheck missing values after conversions and update the dataset info
economic_data_missing_updated = economic_data.isnull().sum()
economic_data_info_updated = economic_data.info()

economic_data_missing_updated, economic_data_info_updated
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 13 columns):
 #   Column
Non-Null Count  Dtype
---  ------
--------------  -----
 0   Time
202 non-null    object
 1   Time Code
200 non-null    object
 2   Country Name
200 non-null    object
 3   Country Code
200 non-null    object
 4   Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]
53 non-null     float64
 5   GDP per capita (current US$) [NY.GDP.PCAP.CD]
194 non-null    float64
 6   GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]
191 non-null    float64
 7   Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
198 non-null    float64
 8   Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]
194 non-null    float64
 9   Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]
185 non-null    float64
 10  Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED
.PC.CD]  186 non-null    float64
 11  Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]
185 non-null    float64
 12  External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]
167 non-null    float64
dtypes: float64(9), object(4)
memory usage: 20.9+ KB
```

Out[3]:

```
(Time
3
 Time Code
5
 Country Name
5
 Country Code
5
 Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]
152
 GDP per capita (current US$) [NY.GDP.PCAP.CD]
11
 GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]
14
 Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
7
 Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]
11
 Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]
20
 Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED.PC.
CD]     19
 Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]
20
 External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]
38
 dtype: int64,
 None)
```

Economic Data Cleaning Update

Updated Missing Values

After cleaning, here's the updated count of missing values per column:

- **Poverty headcount ratio: Significant missing data. Depending on usage, consider if this column is critical.**
- **GDP and Health Expenditure columns: Various levels of missing data, typically around 5-20% missing.**
- **Secure Internet servers and Infant Mortality : Few missing entries.**

Next Steps

- **Impute or Remove Missing Values: For critical numeric fields, we can impute missing values using mean, median, or another strategy. For categorical fields like country code or name, missing entries might need to be dropped if they can't be accurately filled.**
- **Final Review and Clean-up : Ensure columns like "Time" and "Country Name" have consistent formats.**

In [4]:

```python
# Calculate median for numeric columns
median_values = economic_data[numeric_columns].median()


# Fill missing values
economic_data[numeric_columns] = economic_data[numeric_columns].fillna(median_values)

# Drop rows where 'Country Name' or 'Country Code' is missing
economic_data.dropna(subset=['Country Name', 'Country Code'], inplace=True)

final_missing_values_pandas = economic_data.isnull().sum()
final_data_preview_pandas = economic_data.head()

final_missing_values_pandas
```

Out[4]:

```
Time
0
```

```
Time Code
0
Country Name
0
Country Code
0
Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]
0
GDP per capita (current US$) [NY.GDP.PCAP.CD]
0
GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]
0
Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
0
Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]
0
Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]
0
Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED.PC.C
D]     0
Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]
0
External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]
0
dtype: int64
```

In [5]:

```
final_data_preview_pandas
```

Out[5]:

	Time	Time Code	Country Name	Country Code	Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]	GDP per capita (current US$) [NY.GDP.PCAP.CD]	GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]	Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]	Mo infant [SP.DY
0	2020	YR2020	Afghanistan	AFG	0.5	516.866797	-5.364666	34.947962	
1	2020	YR2020	Albania	ALB	0.0	5343.037704	-2.745239	884.825091	
2	2020	YR2020	Algeria	DZA	0.5	3354.157303	-6.729942	48.467647	
3	2020	YR2020	Andorra	AND	0.5	37207.222000	-12.735078	9665.379665	
4	2020	YR2020	Angola	AGO	0.5	1502.950754	-8.672432	19.743640	

Loading economic data to OLTP database

The next step I choose was to load the cleaned data to OLTP so it can be queried during ETL process. Creating database named `olympic_oltp` to store all the data that was cleaned

In [6]:

```
import psycopg2
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT

# Parameters to connect to the default PostgreSQL database
params = {
    'dbname': 'postgres',
    'user': 'postgres',
    'password': 'postgres',
    'host': 'pgdb'
}

try:
    # Connect to the PostgreSQL server
```

```
    conn = psycopg2.connect(**params)

    # Enable autocommit so operations like creating a database are committed without havi
ng to call conn.commit()
    conn.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT)

    # Create a cursor object
    cursor = conn.cursor()

    # Name of the new database
    new_db_name = 'olympic_oltp'  # Replace with the name of the database you want to cre
ate

    # Ensure the database name is safe to use
    # For example, by checking against a list of allowed names or patterns
    if not new_db_name.isidentifier():
        raise ValueError("Invalid database name.")

    # Create a new database using an f-string
    cursor.execute(f"CREATE DATABASE {new_db_name}")

    print("Database created successfully")

    # Close communication with the database
    cursor.close()
    conn.close()

except Exception as e:
    print(f"An error occurred: {e}")
```

```
Database created successfully
```

In [7]:

```
from psycopg2 import OperationalError
def create_connection(db_name, db_user, db_password, db_host, db_port):
    connection = None
    try:
        connection = psycopg2.connect(
            database=db_name,
            user=db_user,
            password=db_password,
            host=db_host,
            port=db_port,
        )
        print("Connection to PostgreSQL DB successful")
    except OperationalError as e:
        print(f"The error '{e}' occurred")
    return connection
```

In [8]:

```
# Connection details
db_name = "olympic_oltp"
db_user = "postgres"
db_password = "postgres"
db_host = "pgdb"
db_port = "5432"

# Create the connection
connection = create_connection(db_name, db_user, db_password, db_host, db_port)
```

```
Connection to PostgreSQL DB successful
```

In [ ]:

```

```

In [9]:

```
# Rename the columns to match the PostgreSQL table schema exactly
```

```
economic_data.columns = [
    'time_year',
    'time_code',
    'country_name',
    'country_code',
    'poverty_ratio',  # Ratio of population at $2.15 a day PPP
    'gdp_per_capita_usd',  # GDP per capita in current US$
    'gdp_per_capita_growth',  # Annual growth of GDP per capita
    'secure_internet_servers_per_million',  # Secure Internet servers per million people
    'infant_mortality_rate',  # Infant mortality rate per 1,000 live births
    'health_expenditure_pct_gdp',  # Current health expenditure as % of GDP
    'gov_health_expenditure_per_capita_usd',  # Government health expenditure per capita
in current US$
    'private_health_expenditure_per_capita_usd',  # Private health expenditure per capita
in current US$
    'external_health_expenditure_per_capita_usd'  # External health expenditure per capit
a in current US$
]
```

In [10]:

```
economic_data.head()
```

Out[10]:

	time_year	time_code	country_name	country_code	poverty_ratio	gdp_per_capita_usd	gdp_per_capita_growth	secure_inter
0	2020	YR2020	Afghanistan	AFG	0.5	516.866797	-5.364666	
1	2020	YR2020	Albania	ALB	0.0	5343.037704	-2.745239	
2	2020	YR2020	Algeria	DZA	0.5	3354.157303	-6.729942	
3	2020	YR2020	Andorra	AND	0.5	37207.222000	-12.735078	
4	2020	YR2020	Angola	AGO	0.5	1502.950754	-8.672432	

In [11]:

```
from sqlalchemy import create_engine

connection_url = f"postgresql://{db_user}:{db_password}@{db_host}:{db_port}/{db_name}"

# Create the engine
engine = create_engine(connection_url)

create_economic_data_table_sql = """
CREATE TABLE IF NOT EXISTS economic_data (
    time_year VARCHAR(255),
    time_code VARCHAR(255),
    country_name VARCHAR(255),
    country_code VARCHAR(255),
    poverty_ratio FLOAT, -- Ratio of population at $2.15 a day PPP
    gdp_per_capita_usd FLOAT, -- GDP per capita in current US$
    gdp_per_capita_growth FLOAT, -- Annual growth of GDP per capita
    secure_internet_servers_per_million FLOAT, -- Secure Internet servers per million peo
ple
    infant_mortality_rate FLOAT, -- Infant mortality rate per 1,000 live births
    health_expenditure_pct_gdp FLOAT, -- Current health expenditure as % of GDP
    gov_health_expenditure_per_capita_usd FLOAT, -- Government health expenditure per cap
ita in current US$
    private_health_expenditure_per_capita_usd FLOAT, -- Private health expenditure per ca
pita in current US$
    external_health_expenditure_per_capita_usd FLOAT -- External health expenditure per c
apita in current US$
);
"""

cursor = connection.cursor()
```

```
cursor.execute(create_economic_data_table_sql)

connection.commit()
cursor.close()
connection.close()
```

I have loaded data into a table named `economic_data` in a SQL database, connecting through `engine` and ensuring that the DataFrame's index is not included as a column in the table.

In [12]:

```
economic_data.to_sql("economic_data", con=engine, if_exists="append", index=False)
```

Out[12]:

```
200
```

In [13]:

```
# Save the cleaned data
economic_data.to_csv('cleaned_data/Economic data.csv', index=False)
```

2. Global Population

The dataset consists of population estimates for various countries from 1980 to 2028, with some entries marked a `no data`. All the columns are currently read as objects (strings), which is typical when dealing with mixed data types like numbers and text.

In [14]:

```
global_population_data = pd.read_csv('./raw_data/Global Population.csv', encoding='ISO-88
59-1')
```

In [15]:

```
global_population_data_info = global_population_data.info()
global_population_data_missing_values = global_population_data.isnull().sum()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 231 entries, 0 to 230
Data columns (total 50 columns):
 #   Column                          Non-Null Count  Dtype 
---  ------                          --------------  ----- 
 0   Population (Millions of people)  229 non-null    object
 1   1980                            228 non-null    object
 2   1981                            228 non-null    object
 3   1982                            228 non-null    object
 4   1983                            228 non-null    object
 5   1984                            228 non-null    object
 6   1985                            228 non-null    object
 7   1986                            228 non-null    object
 8   1987                            228 non-null    object
 9   1988                            228 non-null    object
 10  1989                            228 non-null    object
 11  1990                            228 non-null    object
 12  1991                            228 non-null    object
 13  1992                            228 non-null    object
 14  1993                            228 non-null    object
 15  1994                            228 non-null    object
 16  1995                            228 non-null    object
 17  1996                            228 non-null    object
 18  1997                            228 non-null    object
 19  1998                            228 non-null    object
 20  1999                            228 non-null    object
 21  2000                            228 non-null    object
 22  2001                            228 non-null    object
 23  2002                            228 non-null    object
 24  2003                            228 non-null    object
 25  2004                            228 non-null    object
```

```
 25  2004                                228 non-null    object
 26  2005                                228 non-null    object
 27  2006                                228 non-null    object
 28  2007                                228 non-null    object
 29  2008                                228 non-null    object
 30  2009                                228 non-null    object
 31  2010                                228 non-null    object
 32  2011                                228 non-null    object
 33  2012                                228 non-null    object
 34  2013                                228 non-null    object
 35  2014                                228 non-null    object
 36  2015                                228 non-null    object
 37  2016                                228 non-null    object
 38  2017                                228 non-null    object
 39  2018                                228 non-null    object
 40  2019                                228 non-null    object
 41  2020                                228 non-null    object
 42  2021                                228 non-null    object
 43  2022                                228 non-null    object
 44  2023                                228 non-null    object
 45  2024                                228 non-null    object
 46  2025                                228 non-null    object
 47  2026                                228 non-null    object
 48  2027                                228 non-null    object
 49  2028                                228 non-null    object
dtypes: object(50)
memory usage: 90.4+ KB
```

In [16]:

```
global_population_data_missing_values
```

Out[16]:

```
Population (Millions of people)    2
1980                               3
1981                               3
1982                               3
1983                               3
1984                               3
1985                               3
1986                               3
1987                               3
1988                               3
1989                               3
1990                               3
1991                               3
1992                               3
1993                               3
1994                               3
1995                               3
1996                               3
1997                               3
1998                               3
1999                               3
2000                               3
2001                               3
2002                               3
2003                               3
2004                               3
2005                               3
2006                               3
2007                               3
2008                               3
2009                               3
2010                               3
2011                               3
2012                               3
2013                               3
2014                               3
2015                               3
2016                               3
2017                               3
```

```
2018                        3
2019                        3
2020                        3
2021                        3
2022                        3
2023                        3
2024                        3
2025                        3
2026                        3
2027                        3
2028                        3
dtype: int64
```

In [17]:

```
global_population_data.replace('no data', np.nan, inplace=True)

# Convert data types for all year columns to float
for col in global_population_data.columns[1:]:  # Assuming the first column is Country Na
me
    global_population_data[col] = pd.to_numeric(global_population_data[col], errors='coe
rce')
```

- **Converted Numeric Columns: All year columns now contain floating-point numbers or `NaN` for missing data.**
- **Handled Missing Values: Missing data is now uniformly represented with `NaN`, making it easier to perform aggregations and other data operations.**

In [18]:

```
global_population_data.head()
```

Out[18]:

	Population (Millions of people)	1980	1981	1982	1983	1984	1985	1986	1987	1988	...	2019	2020	2021	2022	20
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	N
1	Afghanistan	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	32.200	32.941	33.698	34.263	N
2	Albania	2.672	2.726	2.784	2.844	2.904	2.965	3.023	3.084	3.142	...	2.881	2.878	2.873	2.866	2.:
3	Algeria	18.666	19.246	19.864	20.516	21.175	22.200	22.800	23.400	24.100	...	43.424	43.851	44.577	45.291	45.!
4	Andorra	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	0.078	0.078	0.080	0.082	0.(

5 rows × 50 columns

Interpolation Method: For population data, `linear interpolation` often makes sense as it assumes a gradual change between points.

In [19]:

```
# Interpolate missing data for each country
global_population_data.iloc[:, 1:] = global_population_data.iloc[:, 1:].apply(lambda x:
x.interpolate(method='linear', limit_direction='both'), axis=1)
```

In [20]:

```
global_population_data = global_population_data.rename(columns={'Population (Millions of
people)': 'Population'})
```

Some rows have all the columns with NaN values which we don't required. We will drop them

In [21]:

```
# Select all columns except the first one for checking NaN values
```

```
condition = global_population_data.iloc[:, 1:].isna().all(axis=1)

# Drop rows based on the condition
global_population_data = global_population_data[~condition]
```

In [22]:

```
global_population_data
```

Out[22]:

	Population	1980	1981	1982	1983	1984	1985	1986	1987	1988	...	2019	
1	Afghanistan	17.887	17.887	17.887	17.887	17.887	17.887	17.887	17.887	17.887	...	32.200	3
2	Albania	2.672	2.726	2.784	2.844	2.904	2.965	3.023	3.084	3.142	...	2.881	
3	Algeria	18.666	19.246	19.864	20.516	21.175	22.200	22.800	23.400	24.100	...	43.424	4
4	Andorra	0.070	0.070	0.070	0.070	0.070	0.070	0.070	0.070	0.070	...	0.078	
5	Angola	8.272	8.495	8.720	8.948	9.185	10.350	10.646	10.918	11.214	...	32.354	3
...	...	...	...	...	...	...	...	...	...	...	...	...	
224	Major advanced economies (G7)	612.155	616.177	619.745	623.047	626.158	629.495	633.018	636.492	640.455	...	767.111	77
225	Middle East and Central Asia	254.673	262.850	271.095	279.549	288.557	297.650	306.682	314.912	323.543	...	822.958	83
226	Other advanced economies	112.560	114.089	115.636	117.012	118.267	119.403	120.510	121.721	123.038	...	173.316	17
227	Sub-Saharan Africa	342.745	352.398	362.565	373.099	384.021	395.902	407.207	418.858	430.534	...	1026.814	105
228	World	4009.286	4082.448	4157.684	4231.389	4306.121	4383.428	4462.501	4542.998	4623.714	...	7593.398	766

228 rows × 50 columns

In [23]:

```
global_population_data.columns
```

Out[23]:

```
Index(['Population', '1980', '1981', '1982', '1983', '1984', '1985', '1986',
       '1987', '1988', '1989', '1990', '1991', '1992', '1993', '1994', '1995',
       '1996', '1997', '1998', '1999', '2000', '2001', '2002', '2003', '2004',
       '2005', '2006', '2007', '2008', '2009', '2010', '2011', '2012', '2013',
       '2014', '2015', '2016', '2017', '2018', '2019', '2020', '2021', '2022',
       '2023', '2024', '2025', '2026', '2027', '2028'],
      dtype='object')
```

In [24]:

```
# Create table using the SQL statement defined above
create_population_data_table_sql = """
CREATE TABLE population_data (
    "Population" TEXT,
    -- Add columns for each year
    "1980" FLOAT, "1981" FLOAT, "1982" FLOAT, "1983" FLOAT, "1984" FLOAT, "1985" FLOAT,
"1986" FLOAT,
    "1987" FLOAT, "1988" FLOAT, "1989" FLOAT, "1990" FLOAT, "1991" FLOAT, "1992" FLOAT,
"1993" FLOAT,
    "1994" FLOAT, "1995" FLOAT, "1996" FLOAT, "1997" FLOAT, "1998" FLOAT, "1999" FLOAT,
"2000" FLOAT,
    "2001" FLOAT, "2002" FLOAT, "2003" FLOAT, "2004" FLOAT, "2005" FLOAT, "2006" FLOAT,
"2007" FLOAT,
```

```
    "2008" FLOAT, "2009" FLOAT, "2010" FLOAT, "2011" FLOAT, "2012" FLOAT, "2013" FLOAT, 
"2014" FLOAT,
    "2015" FLOAT, "2016" FLOAT, "2017" FLOAT, "2018" FLOAT, "2019" FLOAT, "2020" FLOAT, 
"2021" FLOAT,
    "2022" FLOAT, "2023" FLOAT, "2024" FLOAT, "2025" FLOAT, "2026" FLOAT, "2027" FLOAT, 
"2028" FLOAT
);
"""
```

In [25]:

```
connection = create_connection(db_name, db_user, db_password, db_host, db_port)
cursor = connection.cursor()
# Execute the SQL statement
cursor.execute(create_population_data_table_sql)


connection.commit()
cursor.close()
connection.close()
```

```
Connection to PostgreSQL DB successful
```

Loading population data to OLTP for ETL process

In [26]:

```
global_population_data.to_sql("population_data", con=engine, if_exists="append", index=F
alse)
```

Out[26]:

```
228
```

In [27]:

```
# Save the cleaned data
global_population_data.to_csv('cleaned_data/Global Population.csv', index=False)
```

3. Life Expectancy

In [28]:

```
life_expectancy_path = './raw_data/life-expectancy.csv'

life_expectancy_data = pd.read_csv(life_expectancy_path)
```

In [29]:

```
life_expectancy_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20755 entries, 0 to 20754
Data columns (total 4 columns):
 #   Column                                            Non-Null Count  Dtype  
---  ------                                            --------------  -----  
 0   Entity                                            20755 non-null  object 
 1   Code                                              19061 non-null  object 
 2   Year                                              20755 non-null  int64  
 3   Period life expectancy at birth - Sex: all - Age: 0  20755 non-null  float64
dtypes: float64(1), int64(1), object(2)
memory usage: 648.7+ KB
```

In [30]:

```
life_expectancy_data.head()
```

Out[30]:

	Entity	Code	Year	Period life expectancy at birth - Sex: all - Age: 0
0	Afghanistan	AFG	1950	27.7275
1	Afghanistan	AFG	1951	27.9634
2	Afghanistan	AFG	1952	28.4456
3	Afghanistan	AFG	1953	28.9304
4	Afghanistan	AFG	1954	29.2258

In [31]:

```
life_expectancy_data.columns
```

Out[31]:

```
Index(['Entity', 'Code', 'Year',
       'Period life expectancy at birth - Sex: all - Age: 0'],
      dtype='object')
```

In [ ]:

```

```

In [32]:

```
# Renamed the column name to make them consistent across and avoiding large column name

life_expectancy_data = life_expectancy_data.rename(columns={'Period life expectancy at bi
rth - Sex: all - Age: 0': 'life_expectancy', 'Entity':'entity', 'Code': 'country_code', '
Year': 'year'})
```

In [33]:

```
life_expectancy_data.columns
```

Out[33]:

```
Index(['entity', 'country_code', 'year', 'life_expectancy'], dtype='object')
```

In [34]:

```
# Create table using the SQL statement defined above
create_life_expectancy_table_query = """
CREATE TABLE life_expectancy_data (
    entity TEXT,
    country_code VARCHAR(250),
    year INT,
    life_expectancy FLOAT
);
"""

connection = create_connection(db_name, db_user, db_password, db_host, db_port)
cursor = connection.cursor()

# Execute the SQL statement
cursor.execute(create_life_expectancy_table_query)
connection.commit()

cursor.close()
connection.close()
```

```
Connection to PostgreSQL DB successful
```

Loading life expectancy data to OLTP

In [35]:

```
life_expectancy_data.to_sql("life_expectancy_data", con=engine, if_exists="append", index
```

```
=False)
```

Out[35]:

```
755
```

Optional

Because, the data contains some regions which does not have region code, I will create 2 CSV file, one with country and other with regions.

We can split the single source CSV into two separate CSV files:

1. **Countries CSV: This will include entries that have a country code and will retain the original country code in the data.**
2. **Regions CSV: This will include entries that originally did not have a country code. These entries will be identified as regions and will not include any code.**

In [36]:

```
# Filter out countries with country code
countries = life_expectancy_data.dropna(subset=['country_code'])

# Filter out regions (entries without a country code)
regions = life_expectancy_data[life_expectancy_data['country_code'].isnull()].copy()
regions.drop('country_code', axis=1, inplace=True)  # Optionally remove the 'Code' colum
n
```

In [37]:

```
# Save the datasets to new CSV files
countries.to_csv('./cleaned_data/life-expectancy-countries.csv', index=False)
regions.to_csv('./cleaned_data/life-expectancy-regions.csv', index=False)
```

4. Mental Illness**

The `mental-illness.csv` file contains the following columns:

- **Entity: Name of the country or region.**
- **Code: The international standard code for the country; some are missing, indicating regions.**
- **Year: Year of the data.**
- **Several columns related to DALYs (Disability-Adjusted Life Years) for different mental health disorders, all age-standardized and for both sexes.**

In [38]:

```
# Load the CSV file to examine its contents and structure
mental_illness_data_path = './raw_data/mental-illness.csv'
mental_illness_data = pd.read_csv(mental_illness_data_path)

# Display the first few rows of the dataframe and summary of the data
mental_illness_data.head()
```

Out[38]:

	Entity	Code	Year	DALYs from depressive disorders per 100,000 people in, both sexes aged age-standardized	DALYs from schizophrenia per 100,000 people in, both sexes aged age-standardized	DALYs from bipolar disorder per 100,000 people in, both sexes aged age-standardized	DALYs from eating disorders per 100,000 people in, both sexes aged age-standardized	DALYs from anxiety disorders per 100,000 people in, both sexes aged age-standardized
0	Afghanistan	AFG	1990	895.22565	138.24825	147.64412	26.471115	440.33000
1	Afghanistan	AFG	1991	893.88434	137.76122	147.56696	25.548681	439.47202
2	Afghanistan	AFG	1992	892.34973	137.08030	147.13086	24.637949	437.60718

				DALYs from depressive disorders per	DALYs from schizophrenia per 100,000 people	DALYs from bipolar disorder per 100,000	DALYs from eating disorders per 100,000	DALYs from anxiety disorders per 100,000 people
3	Afghanistan	AFG	1993	891.51587	136.48602	146.78812	23.863169	436.69104
4	Afghanistan	AFG	1994	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

In [39]:

```
mental_illness_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6840 entries, 0 to 6839
Data columns (total 8 columns):
 #   Column
Non-Null Count  Dtype
---  ------
--------------  -----
 0   Entity
6840 non-null   object
 1   Code
6150 non-null   object
 2   Year
6840 non-null   int64
 3   DALYs from depressive disorders per 100,000 people in, both sexes aged age-standardi
zed  6840 non-null   float64
 4   DALYs from schizophrenia per 100,000 people in, both sexes aged age-standardized
6840 non-null   float64
 5   DALYs from bipolar disorder per 100,000 people in, both sexes aged age-standardized
6840 non-null   float64
 6   DALYs from eating disorders per 100,000 people in, both sexes aged age-standardized
6840 non-null   float64
 7   DALYs from anxiety disorders per 100,000 people in, both sexes aged age-standardized
6840 non-null   float64
dtypes: float64(5), int64(1), object(2)
memory usage: 427.6+ KB
```

In [40]:

```
mental_illness_data.columns
```

Out[40]:

```
Index(['Entity', 'Code', 'Year',
       'DALYs from depressive disorders per 100,000 people in, both sexes aged age-standa
rdized',
       'DALYs from schizophrenia per 100,000 people in, both sexes aged age-standardized'
,
       'DALYs from bipolar disorder per 100,000 people in, both sexes aged age-standardiz
ed',
       'DALYs from eating disorders per 100,000 people in, both sexes aged age-standardiz
ed',
       'DALYs from anxiety disorders per 100,000 people in, both sexes aged age-standardi
zed'],
      dtype='object')
```

In [41]:

```
mental_illness_data.columns = [
    'entity',
    'country_code',
    'year',
    'daly_depression',
    'daly_schizophrenia',
    'daly_bipolar_disorder',
    'daly_eating_disorder',
    'daly_anxiety'
]
```

In [42]:

```
mental_illness_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6840 entries, 0 to 6839
Data columns (total 8 columns):
```

```
Data columns (total 8 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   entity                6840 non-null   object 
 1   country_code          6150 non-null   object 
 2   year                  6840 non-null   int64  
 3   daly_depression       6840 non-null   float64
 4   daly_schizophrenia    6840 non-null   float64
 5   daly_bipolar_disorder 6840 non-null   float64
 6   daly_eating_disorder  6840 non-null   float64
 7   daly_anxiety          6840 non-null   float64
dtypes: float64(5), int64(1), object(2)
memory usage: 427.6+ KB
```

In [43]:

```
create_mental_illness_table_query = """
    CREATE TABLE mental_health_data (
        entity TEXT,
        country_code VARCHAR(250),
        year INT,
        daly_depression FLOAT,
        daly_schizophrenia FLOAT,
        daly_bipolar_disorder FLOAT,
        daly_eating_disorder FLOAT,
        daly_anxiety FLOAT
    );
    """
```

In [44]:

```
connection = create_connection(db_name, db_user, db_password, db_host, db_port)
cursor = connection.cursor()

cursor.execute(create_mental_illness_table_query)

connection.commit()
cursor.close()
connection.close()
```

```
Connection to PostgreSQL DB successful
```

In [45]:

```
mental_illness_data.to_sql("mental_health_data", con=engine, if_exists="append", index=Fa
lse)
```

Out[45]:

```
840
```

1. **Countries CSV: This file will contain entries with country codes. It will retain the original country code.**
2. **Regions CSV: This file will contain entries that originally did not have a country code. These entries will be identified as regions, and the 'Code' column will be dropped.**

In [46]:

```
# Filter out countries with country code
countries_mental = mental_illness_data.dropna(subset=['country_code'])

# Filter out regions (entries without a country code)
regions_mental = mental_illness_data[mental_illness_data['country_code'].isnull()].copy()
regions_mental.drop('country_code', axis=1, inplace=True)  # Remove the 'country_code' co
lumn

# Save the datasets to new CSV files
countries_mental_file = './cleaned_data/mental-illness-countries.csv'
regions_mental_file = './cleaned_data/mental-illness-regions.csv'
countries_mental.to_csv(countries_mental_file, index=False)
regions_mental.to_csv(regions_mental_file, index=False)
```

```
countries_mental_file, regions_mental_file
```

Out[46]:

```
('./cleaned_data/mental-illness-countries.csv',
 './cleaned_data/mental-illness-regions.csv')
```

5. Olympic Hosts**

The `olympic_hosts.csv` file contains information about various Olympic Games, and from the initial inspection, the data appears well-structured with the following columns:

- **game_slug: A unique identifier for each Olympic event.**
- **game_end_date: The end date of the event in an ISO 8601 format.**
- **game_start_date: The start date of the event in an ISO 8601 format.**
- **game_location: The location (country) of the event.**
- **game_name: The name of the Olympic event.**
- **game_season: Specifies whether the games are Summer or Winter Olympics.**
- **game_year: The year the games were held.**

Observations and Possible Data Cleaning Steps:

1. **Date Format: The start and end dates are in ISO 8601 format with time components.**
    - **Convert these date strings to a standard Python `datetime` object for easier manipulation and extraction of specific date components (e.g., just the date without the time).**
    - **Extract just the date part if the time component is not relevant.**
2. **Consistency in Game Location Names: It might be helpful to ensure that all entries under `game_location` are consistently formatted or spelled, particularly for countries that might have undergone name changes or different spelling conventions over the years.**

Example Cleaning Process:

cleaning the date formats by converting the start and end dates to just include the date part, ensuring we have consistent datetime formats. (Optional). Left here for reference.

```python
from datetime import datetime

# Convert date columns to datetime
olympic_hosts_data['game_start_date'] = pd.to_datetime(olympic_hosts_data['game_start_date']).dt.date
olympic_hosts_data['game_end_date'] = pd.to_datetime(olympic_hosts_data['game_end_date']).dt.date

# Example of checking for consistent formatting in 'game_location'
print(olympic_hosts_data['game_location'].unique())
```

In [47]:

```python
olympic_host_data_path = './raw_data/olympic_hosts.csv'

olympic_host_data = pd.read_csv(olympic_host_data_path)
```

In [48]:

```python
olympic_host_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53 entries, 0 to 52
Data columns (total 7 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   game_slug        53 non-null     object
```

```
 1   game_end_date    53 non-null     object
 2   game_start_date  53 non-null     object
 3   game_location    53 non-null     object
 4   game_name        53 non-null     object
 5   game_season      53 non-null     object
 6   game_year        53 non-null     int64
dtypes: int64(1), object(6)
memory usage: 3.0+ KB
```

In [49]:

```
olympic_host_data.columns
```

Out[49]:

```
Index(['game_slug', 'game_end_date', 'game_start_date', 'game_location',
       'game_name', 'game_season', 'game_year'],
      dtype='object')
```

In [50]:

```
from datetime import datetime

# Convert date columns to datetime
olympic_host_data['game_start_date'] = pd.to_datetime(olympic_host_data['game_start_date
']).dt.date
olympic_host_data['game_end_date'] = pd.to_datetime(olympic_host_data['game_end_date']).
dt.date
```

In [51]:

```
olympic_host_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53 entries, 0 to 52
Data columns (total 7 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   game_slug        53 non-null     object
 1   game_end_date    53 non-null     object
 2   game_start_date  53 non-null     object
 3   game_location    53 non-null     object
 4   game_name        53 non-null     object
 5   game_season      53 non-null     object
 6   game_year        53 non-null     int64
dtypes: int64(1), object(6)
memory usage: 3.0+ KB
```

In [52]:

```
create_olympic_hosts_table_query = """
CREATE TABLE olympic_hosts (
    game_slug TEXT,
    game_end_date DATE,
    game_start_date DATE,
    game_location TEXT,
    game_name TEXT,
    game_season TEXT,
    game_year INT
);
"""
```

In [53]:

```
connection = create_connection(db_name, db_user, db_password, db_host, db_port)
cursor = connection.cursor()

cursor.execute(create_olympic_hosts_table_query)
connection.commit()  # Commit the changes

cursor.close()
```

```
connection.close()
```

```
Connection to PostgreSQL DB successful
```

In [54]:

```
olympic_host_data.to_sql("olympic_hosts", con=engine, if_exists="append", index=False)
```

Out[54]:

```
53
```

6. Olympic Medals

In [55]:

```
# Load the newly uploaded CSV file to examine its contents and structure
olympic_medals_data_path = './raw_data/olympic_medals.csv'
olympic_medals_data = pd.read_csv(olympic_medals_data_path)
```

In [56]:

```
olympic_medals_data.head()
```

Out[56]:

	discipline_title	slug_game	event_title	event_gender	medal_type	participant_type	participant_title	
0	Curling	beijing-2022	Mixed Doubles	Mixed	GOLD	GameTeam	Italy	https://olympics.com/en
1	Curling	beijing-2022	Mixed Doubles	Mixed	GOLD	GameTeam	Italy	https://olympics.com/
2	Curling	beijing-2022	Mixed Doubles	Mixed	SILVER	GameTeam	Norway	https://olympics.com/e
3	Curling	beijing-2022	Mixed Doubles	Mixed	SILVER	GameTeam	Norway	https://olympics.com/en
4	Curling	beijing-2022	Mixed Doubles	Mixed	BRONZE	GameTeam	Sweden	https://olympics.com/e

In [57]:

```
olympic_medals_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21697 entries, 0 to 21696
Data columns (total 12 columns):
 #   Column                  Non-Null Count  Dtype 
---  ------                  --------------  ----- 
 0   discipline_title        21697 non-null  object
 1   slug_game               21697 non-null  object
 2   event_title             21697 non-null  object
 3   event_gender            21697 non-null  object
 4   medal_type              21697 non-null  object
 5   participant_type        21697 non-null  object
 6   participant_title       6584 non-null   object
 7   athlete_url             17027 non-null  object
 8   athlete_full_name       18073 non-null  object
 9   country_name            21697 non-null  object
 10  country_code            20195 non-null  object
 11  country_3_letter_code   21697 non-null  object
dtypes: object(12)
memory usage: 2.0+ MB
```

In [58]:

```
olympic_medals_data_missing_values = olympic_medals_data.isnull().sum()
```

In [59]:

```
olympic_medals_data_missing_values
```

Out[59]:

```
discipline_title            0
slug_game                   0
event_title                 0
event_gender                0
medal_type                  0
participant_type            0
participant_title       15113
athlete_url              4670
athlete_full_name        3624
country_name                0
country_code             1502
country_3_letter_code       0
dtype: int64
```

Cleaning Steps

1. **Missing Values:**

   - **Participant Title**: Only team event has participant title. Seperate dimension
   - **Athlete URL & Full Name**: Only individual event has athlete name. Seperate dimension with athlete and url
   - **Country Code:** Drop `country_name` as they are not quite standard. 3 character will be used for consistency

2. **Consistency and Accuracy:**

   - **Check for Consistent Capitalization**: Columns like `country_name`, `event_title`, and `athlete_full_name` should have consistent capitalization to avoid duplications due to case differences.
   - **Validate Country Codes:** Ensure that `country_code` and `country_3_letter_code` are consistent and correctly mapped to `country_name`.

In [60]:

```
create_olympic_medals_table_query = """
CREATE TABLE olympic_medals (
    discipline_title TEXT,
    slug_game TEXT,
    event_title TEXT,
    event_gender VARCHAR(250),
    medal_type TEXT,
    participant_type TEXT,
    participant_title TEXT,
    athlete_url TEXT,
    athlete_full_name TEXT,
    country_name TEXT,
    country_code VARCHAR(10),
    country_3_letter_code VARCHAR(10)
);
"""
```

In [61]:

```
connection = create_connection(db_name, db_user, db_password, db_host, db_port)
cursor = connection.cursor()
```

```
Connection to PostgreSQL DB successful
```

In [62]:

```
cursor.execute(create_olympic_medals_table_query)
connection.commit()   # Commit the changes
cursor.close()
```

```
connection.close()
```

In [63]:

```
olympic_medals_data.to_sql("olympic_medals", con=engine, if_exists="append", index=False)
```

Out[63]:

```
697
```

7. List of countries

Since, there are lot of name mismatch, I used standard names from https://en.wikipedia.org/wiki/List_of_ISO_3166_country_codes standard names from wikipedia

I loaded the external data which has details about names and continents from the url below. https://statisticstimes.com/geography/countries-by-continents.php

After downloading the json from the source (https://statisticstimes.com/m/geography/json/countries-continents.json) above, I loaded them into OLTP database.

In [64]:

```
# Read the JSON file into a pandas DataFrame
countries_data_path = './raw_data/countries-continents.json'
countries_df = pd.read_json(countries_data_path)

# Display the DataFrame to confirm the contents
countries_df.head()
```

Out[64]:

	id	name	M49 code	ISO alpha3 code	continent	region	color
0	AF	Afghanistan	4	AFG	Asia	Southern Asia	#00CC99
1	AX	Åland Islands	248	ALA	Europe	Northern Europe	#99CCFF
2	AL	Albania	8	ALB	Europe	Southern Europe	#8AB8E6
3	DZ	Algeria	12	DZA	Africa	Northern Africa	#2EB82E
4	AS	American Samoa	16	ASM	Oceania	Polynesia	#B88A00

In [65]:

```
countries_df = countries_df.drop(['color', 'id', 'M49 code', 'region'], axis=1)
```

In [66]:

```
countries_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 248 entries, 0 to 247
Data columns (total 3 columns):
 #   Column           Non-Null Count  Dtype 
---  ------           --------------  ----- 
 0   name             248 non-null    object
 1   ISO alpha3 code  248 non-null    object
 2   continent        248 non-null    object
dtypes: object(3)
memory usage: 5.9+ KB
```

In [67]:

```
countries_df.columns = [ "country_name", "country_code", "continent"]
```

In [68]:

```
countries_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 248 entries, 0 to 247
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   country_name  248 non-null    object
 1   country_code  248 non-null    object
 2   continent     248 non-null    object
dtypes: object(3)
memory usage: 5.9+ KB
```

In [69]:

```
create_countries_table_query = """
CREATE TABLE countries (
   country_code CHAR(3) PRIMARY KEY,
   country_name VARCHAR(250) NOT NULL,
   continent VARCHAR(250) NOT NULL
);
"""
```

In [70]:

```
connection = create_connection(db_name, db_user, db_password, db_host, db_port)
cursor = connection.cursor()
```

```
Connection to PostgreSQL DB successful
```

In [71]:

```
cursor.execute(create_countries_table_query)
connection.commit()   # Commit the changes
cursor.close()
connection.close()
```

In [72]:

```
countries_df.to_sql("countries", con=engine, if_exists="append", index=False)
```

Out[72]:

```
248
```

In [ ]:

```

```

ETL (Extract, Transform, Load)

In [1]:

```
import pandas as pd
import numpy as np
```

I created `olympic_olap` database to create dimension table and fact tables here.

In [2]:

```
import psycopg2
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT

# Parameters to connect to the default PostgreSQL database
params = {
    'dbname': 'postgres',
    'user': 'postgres',
    'password': 'postgres',
    'host': 'pgdb'
}

try:
    # Connect to the PostgreSQL server
    conn = psycopg2.connect(**params)

    # Enable autocommit so operations like creating a database are committed without havi
ng to call conn.commit()
    conn.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT)

    # Create a cursor object
    cursor = conn.cursor()

    # Name of the new database
    new_db_name = 'olympic_olap'  # Replace with the name of the database you want to cre
ate

    # Ensure the database name is safe to use
    if not new_db_name.isidentifier():
        raise ValueError("Invalid database name.")

    # Create a new database using an f-string
    cursor.execute(f"CREATE DATABASE {new_db_name}")

    print("Database created successfully")

    # Close communication with the database
    cursor.close()
    conn.close()

except Exception as e:
    print(f"An error occurred: {e}")
```

```
Database created successfully
```

In [3]:

```
from psycopg2 import OperationalError
def create_connection(db_name, db_user, db_password, db_host, db_port):
    connection = None
    try:
        connection = psycopg2.connect(
            database=db_name,
            user=db_user,
            password=db_password,
            host=db_host,
```

```
            port=db_port,
        )
        print("Connection to PostgreSQL DB successful")
    except OperationalError as e:
        print(f"The error '{e}' occurred")
    return connection
```

In [4]:

```
# Connection details
olap_db_name = "olympic_olap"
db_user = "postgres"
db_password = "postgres"
db_host = "pgdb"
db_port = "5432"
```

In [5]:

```
olap_connection = create_connection(olap_db_name, db_user, db_password, db_host, db_port
)
olap_cursor = olap_connection.cursor()
```

```
Connection to PostgreSQL DB successful
```

SQL Statement for Dimension and Fact tables

In [6]:

```
# Create DimLocation

olap_cursor.execute("""
CREATE TABLE DimLocation (
    country_code CHAR(3) NOT NULL PRIMARY KEY,
    country_name VARCHAR(255),
    continent VARCHAR(255)
);
""")
```

In [7]:

```
# Create DimEvent table

olap_cursor.execute("""
CREATE TABLE DimEvent (
    event_id SERIAL PRIMARY KEY,
    event_title VARCHAR(250),
    event_discipline VARCHAR(250),
    event_gender VARCHAR(250)
);
""")
```

In [8]:

```
# Create DimParticipant table

olap_cursor.execute("""
CREATE TABLE DimParticipant (
    participant_id SERIAL PRIMARY KEY,
    participant_title VARCHAR(255),
    participant_type VARCHAR(100)
);
""")
```

In [9]:

```
# Create DimAthlete table

olap_cursor.execute("""
CREATE TABLE DimAthlete (
```

```
    athlete_id SERIAL PRIMARY KEY,
    athlete_name VARCHAR(250),
    athlete_url VARCHAR(250)
);
""")
```

In [10]:

```
# Create DimYear table

olap_cursor.execute("""
CREATE TABLE DimYear (
    year INTEGER NOT NULL PRIMARY KEY
);
""")
```

In [11]:

```
# Create DimGame table

olap_cursor.execute("""
CREATE TABLE DimGame (
    game_slug VARCHAR(100) NOT NULL PRIMARY KEY,
    game_name VARCHAR(100),
    game_season VARCHAR(10),
    game_year INTEGER,
    country_code CHAR(3)
);
""")
```

In [12]:

```
# Create FactOlympicMedalsMeasures

olap_cursor.execute("""
CREATE TABLE FactOlympicMedalsMeasures (
    game_slug VARCHAR(100) REFERENCES DimGame(game_slug),
    participant_id INTEGER REFERENCES DimParticipant(participant_id),
    athlete_id INTEGER REFERENCES DimAthlete(athlete_id),
    event_id INTEGER REFERENCES DimEvent(event_id),
    country_code CHAR(3) NOT NULL REFERENCES DimLocation(country_code),
    year INTEGER NOT NULL REFERENCES DimYear(year),
    total_bronze_medals INTEGER,
    total_silver_medals INTEGER,
    total_gold_medals INTEGER,
    total_medals INTEGER
);
""")
```

In [13]:

```
# Create FactEconomicMeasure

olap_cursor.execute("""
CREATE TABLE FactEconomicMeasure (
    year INTEGER NOT NULL REFERENCES DimYear(year),
    country_code CHAR(3) NOT NULL REFERENCES DimLocation(country_code),
    poverty_count FLOAT,
    gdp_per_capita FLOAT,
    annual_gdp_growth FLOAT,
    servers_count INTEGER
);
""")
```

In [14]:

```
# Create FactHealthMeasure

olap_cursor.execute("""
CREATE TABLE FactHealthMeasure (
    year INTEGER NOT NULL REFERENCES DimYear(year),
```

```
    country_code CHAR(3) NOT NULL REFERENCES DimLocation(country_code),
    daly_depression FLOAT,
    daly_schizophrenia FLOAT,
    daly_bipolar_disorder FLOAT,
    daly_eating_disorder FLOAT,
    daly_anxiety FLOAT,
    life_expectancy FLOAT,
    infant_mortality_rate FLOAT,
    current_health_expenditure FLOAT,
    government_health_expenditure FLOAT,
    private_health_expenditure FLOAT,
    external_health_expenditure FLOAT
);
""")
```

Loading data to Dimension tables

1. DimYear

In [15]:

```
# Insert values from 1896 - start of the olympic data to 2022

olap_cursor.execute("""
INSERT INTO DimYear (year)
SELECT generate_series AS year
FROM generate_series(1896, 2022);
""")

olap_connection.commit()
```

2. DimEvent

Loading data to DimensionEvent. For this I have chosen `event_title`, `discipline_title` and `event_gender` so that these things can be represented by `event_id` in the fact table.

In [16]:

```
oltp_connection = create_connection('olympic_oltp', db_user, db_password, db_host, db_po
rt)

oltp_cursor = oltp_connection.cursor()
```

```
Connection to PostgreSQL DB successful
```

In [17]:

```
oltp_cursor.execute("""
SELECT DISTINCT
    event_title,
    discipline_title AS event_discipline,
    event_gender
FROM
    olympic_medals
ORDER BY
    event_title, event_gender;
""")

# Fetch the data from database
dim_event_data = oltp_cursor.fetchall()

# all column names
dim_event_data_columns = [desc[0] for desc in oltp_cursor.description]

# Create dataframe with corresponding column names
dim_event_df = pd.DataFrame(dim_event_data, columns = dim_event_data_columns)
```

```
dim_event_df
```

Out[17]:

	event_title	event_discipline	event_gender
0	0.5-1t mixed	Sailing	Open
1	0.5t mixed, race one	Sailing	Open
2	0.5t mixed, race two	Sailing	Open
3	10000m men	Speed skating	Men
4	10000m men	Athletics	Men
...	...	...	...
1583	Women's Uneven Bars	Artistic Gymnastics	Women
1584	Women's Vault	Artistic Gymnastics	Women
1585	Women's Welter (64-69kg)	Boxing	Women
1586	Yngling - Keelboat women	Sailing	Women
1587	york round (100y - 80y - 60y) men	Archery	Men

1588 rows × 3 columns

In [18]:

```
from sqlalchemy import create_engine

olap_connection_url = f"postgresql://{db_user}:{db_password}@{db_host}:{db_port}/{olap_db_name}"

# Create the engine
olap_engine = create_engine(olap_connection_url)


# Load the dataframe to DimEvent table
dim_event_df.to_sql("dimevent", con=olap_engine, if_exists="append", index=False)
```

Out[18]:

```
588
```

3. DimParticipant

In [19]:

```
oltp_cursor.execute("""
SELECT DISTINCT
    participant_title,
    participant_type
FROM
    olympic_medals;
""")

dim_participant_data = oltp_cursor.fetchall()
```

In [20]:

```
dim_participant_data_columns = [desc[0] for desc in oltp_cursor.description]
dim_participant_data_columns
```

Out[20]:

```
['participant_title', 'participant_type']
```

In [21]:

```
dim_participant_df = pd.DataFrame(dim_participant_data, columns = dim_participant_data_co
```

```
lumns)
dim_participant_df.head()
```

Out[21]:

	participant_title	participant_type
0	Romania team	GameTeam
1	Tan-Fe-Pah	GameTeam
2	Independent Rowing Club #3	GameTeam
3	Elsie	GameTeam
4	Sans Atout #1	GameTeam

In [22]:

```
dim_participant_df.to_sql("dimparticipant", con=olap_engine, if_exists="append", index=Fa
lse)
```

Out[22]:

```
494
```

4. DimAthlete

In [23]:

```
oltp_cursor.execute("""
SELECT DISTINCT
    athlete_full_name as athlete_name,
    athlete_url
FROM
    olympic_medals
WHERE
    athlete_full_name IS NOT NULL AND athlete_url IS NOT NULL
ORDER BY athlete_name;
""")

dim_athlete_data = oltp_cursor.fetchall()
```

In [24]:

```
dim_athlete_data_columns = [desc[0] for desc in oltp_cursor.description]
dim_athlete_data_columns
```

Out[24]:

```
['athlete_name', 'athlete_url']
```

In [25]:

```
dim_athlete_df = pd.DataFrame(dim_athlete_data, columns = dim_athlete_data_columns)
dim_athlete_df.head()
```

Out[25]:

	athlete_name	athlete_url
0	Aage Ernst LARSEN	https://olympics.com/en/athletes/aage-ernst-la...
1	Aage Ingvar ERIKSEN	https://olympics.com/en/athletes/aage-ingvar-e...
2	Aagje Ada KOK	https://olympics.com/en/athletes/aagje-ada-kok
3	Aarne Eemeli REINI	https://olympics.com/en/athletes/aarne-eemeli-...
4	Aaron CHIA	https://olympics.com/en/athletes/aaron-chia

In [26]:

```
dim_athlete_df.to_sql("dimathlete", con=olap_engine, if_exists="append", index=False)
```

```
dim_athlete_df.to_sql("dimathlete", con=olap_engine, if_exists="append", index=False)
```

Out[26]:

```
116
```

5. DimLocation

This is the most extensive part of the ETL process. There are lot of names that does not match with each other. I have used standard names and new country codes instead which has been explained in the OLTP notebook. From there, the idea is to use the same standard names everywhere. So, i will replace the non-standard names with the one in the standard table. This will make the process more streamline and avoid data duplication and deletion. This analysis and comparisons have been documented here.

https://uniwa-my.sharepoint.com/:x:/g/personal/23771397_student_uwa_edu_au/EVQc_vWogmVChmgQyyvDT0wBvB6OIdA7985e=BBcXYv

In [27]:

```
oltp_cursor.execute("""
SELECT * FROM countries;
""")

dim_location_data = oltp_cursor.fetchall()
```

In [28]:

```
dim_location_data_columns = [desc[0] for desc in oltp_cursor.description]
dim_location_data_columns
```

Out[28]:

```
['country_code', 'country_name', 'continent']
```

In [29]:

```
dim_location_df = pd.DataFrame(dim_location_data, columns = dim_location_data_columns)
dim_location_df.head()
```

Out[29]:

	country_code	country_name	continent
0	AFG	Afghanistan	Asia
1	ALA	Åland Islands	Europe
2	ALB	Albania	Europe
3	DZA	Algeria	Africa
4	ASM	American Samoa	Oceania

In [30]:

```
dim_location_df.to_sql("dimlocation", con=olap_engine, if_exists="append", index=False)
```

Out[30]:

```
248
```

I played around with the country name running query like the one to see the name discrepancies.

```
SELECT DISTINCT o.game_location
FROM olympic_hosts o
LEFT JOIN countries c ON o.game_location = c.country_name
WHERE c.country_name IS NULL;
```

Running above code we find that some countries are named differently in olympic_host file

"Australia, Sweden" -> "Australia" \ "Federal Republic of Germany" -> "Germany" "Great Britain" -> United Kingdom of Great Britain and Northern Ireland \ "United States" -> United States of America \ "USSR" -> Russian Federation \ "Yugoslavia" -> Serbia

In [31]:

```
oltp_cursor.execute("""
CREATE TABLE olympic_hosts_backup AS
SELECT *
FROM olympic_hosts;
""")

oltp_connection.commit()
```

In [32]:

```
oltp_cursor.execute("""
UPDATE olympic_hosts
SET game_location = CASE
    WHEN game_location = 'Australia, Sweden' THEN 'Australia'
    WHEN game_location = 'Federal Republic of Germany' THEN 'Germany'
    WHEN game_location = 'Great Britain' THEN 'United Kingdom of Great Britain and Northe
rn Ireland'
    WHEN game_location = 'United States' THEN 'United States of America'
    WHEN game_location = 'USSR' THEN 'Russian Federation'
    WHEN game_location = 'Yugoslavia' THEN 'Serbia'
    ELSE game_location
END;
""")

oltp_connection.commit()
```

In [33]:

```
oltp_cursor.execute("""
CREATE TABLE olympic_medals_backup AS
SELECT *
FROM olympic_medals;
""")

oltp_connection.commit()
```

The countries appear in almost all the CSV file. I had to create a standard table to make it consistent across the database. Following is the sample of table I used. The more details are in the link provided above.

Old Code	New Code	Country Name	Remarks
AHO	NLD	Netherlands Antilles	Dissolved in 2010
ALG	DZA	Algeria	
ANZ	None	Australia and New Zealand	Historical context, no single current ISO code
BAH	BHS	Bahamas	
BAR	BRB	Barbados	
BER	BMU	Bermuda	
BOH	CZE	Bohemia	Historical region, now part of Czech Republic
BOT	BWA	Botswana	
BUL	BGR	Bulgaria	
BUR	MMR	Burma	Now Myanmar
CHI	CHL	Chile	
CRC	CRI	Costa Rica	
CRO	HRV	Croatia	

Old Code	New Code	Country Name	Remarks
DEN	DNK	Denmark	
EUN	None	Unified Team	Represented former Soviet Union republics in 1992
FIJ	FJI	Fiji	
FRG	DEU	Federal Republic of Germany	Now Germany
GDR	DEU	German Democratic Republic	Now part of Germany
GER	DEU	Germany	
GRE	GRC	Greece	
GRN	GRD	Grenada	
GUA	GTM	Guatemala	
HAI	HTI	Haiti	
INA	IDN	Indonesia	
IOA	None	Independent Olympic Athletes	No standard ISO code
IRI	IRN	Iran	
ISV	VIR	Virgin Islands, U.S.	
KOS	XKX	Kosovo	Not universally recognized
KSA	SAU	Saudi Arabia	
KUW	KWT	Kuwait	
LAT	LVA	Latvia	
MAS	MYS	Malaysia	
MGL	MNG	Mongolia	
MIX	None	Mixed team	No standard ISO code
MRI	MUS	Mauritius	
NED	NLD	Netherlands	
NGR	NGA	Nigeria	
NIG	NER	Niger	
OAR	None	Olympic Athletes from Russia	No standard ISO code
PAR	PRY	Paraguay	
PHI	PHL	Philippines	
POR	PRT	Portugal	
PUR	PRI	Puerto Rico	
ROC	TWN	Taiwan, Republic of China	Commonly used ISO code is TWN for Taiwan
RSA	ZAF	South Africa	
SAM	WSM	Samoa	
SCG	SRB/MNE	Serbia and Montenegro	Dissolved, now Serbia SRB and Montenegro MNE - Will chose SRB
SLO	SVN	Slovenia	
SRI	LKA	Sri Lanka	
SUD	SDN	Sudan	
SUI	CHE	Switzerland	
TAN	TZA	Tanzania	
TCH	CZE/SVK	Czechoslovakia	Now Czech Republic CZE, and Slovakia SVK - Will Choose CZE
TGA	TON	Tonga	
TOG	TGO	Togo	

Old Code	New Code	Country Name	Remarks
TPE	TWN	Chinese Taipei	Commonly used ISO code is TWN for Taiwan
UAE	ARE	United Arab Emirates	
UAR	EGY	United Arab Republic	Dissolved, was a union between Egypt and Syria
URS	RUS	Soviet Union	Dissolved, the largest successor state is Russia
URU	URY	Uruguay	
VIE	VNM	Vietnam	
WIF	None	West Indies Federation	Dissolved, was a political union of Caribbean islands
YUG	SRB/HRV	Yugoslavia	Dissolved, successor states include Serbia SRB, Croatia HRV, etc. - Will choose SRB
ZAM	ZMB	Zambia	
ZIM	ZWE	Zimbabwe	

I applied update to the `olympic_medals` first replacing the old code with new country code

In [34]:

```
oltp_cursor.execute("""
-- Applying updates for each old code to the new code
UPDATE olympic_medals SET country_3_letter_code = CASE
    WHEN country_3_letter_code = 'AHO' THEN 'NLD'
    WHEN country_3_letter_code = 'ALG' THEN 'DZA'
    WHEN country_3_letter_code = 'BAH' THEN 'BHS'
    WHEN country_3_letter_code = 'BAR' THEN 'BRB'
    WHEN country_3_letter_code = 'BER' THEN 'BMU'
    WHEN country_3_letter_code = 'BOH' THEN 'CZE'
    WHEN country_3_letter_code = 'BOT' THEN 'BWA'
    WHEN country_3_letter_code = 'BUL' THEN 'BGR'
    WHEN country_3_letter_code = 'BUR' THEN 'MMR'
    WHEN country_3_letter_code = 'CHI' THEN 'CHL'
    WHEN country_3_letter_code = 'CRC' THEN 'CRI'
    WHEN country_3_letter_code = 'CRO' THEN 'HRV'
    WHEN country_3_letter_code = 'DEN' THEN 'DNK'
    WHEN country_3_letter_code = 'FIJ' THEN 'FJI'
    WHEN country_3_letter_code = 'FRG' THEN 'DEU'
    WHEN country_3_letter_code = 'GDR' THEN 'DEU'
    WHEN country_3_letter_code = 'GER' THEN 'DEU'
    WHEN country_3_letter_code = 'GRE' THEN 'GRC'
    WHEN country_3_letter_code = 'GRN' THEN 'GRD'
    WHEN country_3_letter_code = 'GUA' THEN 'GTM'
    WHEN country_3_letter_code = 'HAI' THEN 'HTI'
    WHEN country_3_letter_code = 'INA' THEN 'IDN'
    WHEN country_3_letter_code = 'IRI' THEN 'IRN'
    WHEN country_3_letter_code = 'ISV' THEN 'VIR'
    WHEN country_3_letter_code = 'KOS' THEN 'XKX'
    WHEN country_3_letter_code = 'KSA' THEN 'SAU'
    WHEN country_3_letter_code = 'KUW' THEN 'KWT'
    WHEN country_3_letter_code = 'LAT' THEN 'LVA'
    WHEN country_3_letter_code = 'MAS' THEN 'MYS'
    WHEN country_3_letter_code = 'MGL' THEN 'MNG'
    WHEN country_3_letter_code = 'MRI' THEN 'MUS'
    WHEN country_3_letter_code = 'NED' THEN 'NLD'
    WHEN country_3_letter_code = 'NGR' THEN 'NGA'
    WHEN country_3_letter_code = 'NIG' THEN 'NER'
    WHEN country_3_letter_code = 'PAR' THEN 'PRY'
    WHEN country_3_letter_code = 'PHI' THEN 'PHL'
    WHEN country_3_letter_code = 'POR' THEN 'PRT'
    WHEN country_3_letter_code = 'PUR' THEN 'PRI'
    WHEN country_3_letter_code = 'ROC' THEN 'TWN'
    WHEN country_3_letter_code = 'RSA' THEN 'ZAF'
    WHEN country_3_letter_code = 'SAM' THEN 'WSM'
    WHEN country_3_letter_code = 'SCG' THEN 'SRB'
    WHEN country_3_letter_code = 'SLO' THEN 'SVN'
    WHEN country_3_letter_code = 'SRI' THEN 'LKA'
    WHEN country_3_letter_code = 'SUD' THEN 'SDN'
    WHEN country_3_letter_code = 'SUI' THEN 'CHE'
```

```
    WHEN country_3_letter_code = 'TAN' THEN 'TZA'
    WHEN country_3_letter_code = 'TCH' THEN 'CZE'
    WHEN country_3_letter_code = 'TGA' THEN 'TON'
    WHEN country_3_letter_code = 'TOG' THEN 'TGO'
    WHEN country_3_letter_code = 'TPE' THEN 'TWN'
    WHEN country_3_letter_code = 'UAE' THEN 'ARE'
    WHEN country_3_letter_code = 'UAR' THEN 'EGY'
    WHEN country_3_letter_code = 'URS' THEN 'RUS'
    WHEN country_3_letter_code = 'URU' THEN 'URY'
    WHEN country_3_letter_code = 'VIE' THEN 'VNM'
    WHEN country_3_letter_code = 'YUG' THEN 'SRB'
    WHEN country_3_letter_code = 'ZAM' THEN 'ZMB'
    WHEN country_3_letter_code = 'ZIM' THEN 'ZWE'
    ELSE country_3_letter_code
END;
""")
```

In [35]:

```
oltp_connection.commit()
```

After performing join operations to see other mismatches, i found the following which has been updated.

In [36]:

```
oltp_cursor.execute("""
UPDATE olympic_medals SET country_3_letter_code = CASE
    WHEN country_3_letter_code = 'EUN' THEN 'RUS'
    WHEN country_3_letter_code = 'OAR' THEN 'RUS'
    ELSE country_3_letter_code
END
""")

oltp_connection.commit()
```

I decided to ignore the following codes as they are ambiguous and does not contribute much to our analysis.

In [37]:

```
oltp_cursor.execute("""
DELETE FROM olympic_medals
WHERE country_3_letter_code IN ('ANZ', 'IOA', 'MIX', 'WIF', 'XKX');
""")


oltp_connection.commit()
```

Running another SQL commands made me realized the below.

```
SELECT DISTINCT o.country_code
FROM life_expectancy_data o
LEFT JOIN countries c ON o.country_code = c.country_code
WHERE c.country_code IS NULL;
```

gave following

Action:

"OWID_WRL" -> world -> remove it \ "OWID_KOS" -> kosovo - not recognised -> \ "OWID_USS -> RUS"

Before performing any operation, I started creating backup first.

a. life expectancy

Replaced the mismatched country code in life expectancy

In [38]:

```
oltp_cursor.execute("""
CREATE TABLE life_expectancy_data_backup AS
SELECT *
FROM life_expectancy_data;
""")

oltp_connection.commit()
```

In [39]:

```
oltp_cursor.execute("""
-- Delete rows with "OWID_WRL" and "OWID_KOS" codes
DELETE FROM life_expectancy_data
WHERE country_code = 'OWID_WRL' OR country_code = 'OWID_KOS';
""")

oltp_cursor.execute("""
DELETE FROM life_expectancy_data
WHERE country_code IS NULL;
""")

oltp_cursor.execute("""
UPDATE life_expectancy_data
SET country_code = 'RUS'
WHERE country_code = 'OWID_USS';
""")

oltp_connection.commit()
```

b. mental health data

Removed `OWID_WRL` as it represented world which is not required as our analysis is based on individual countries. We can always sum up all the data for each country to find the world data.

In [40]:

```
oltp_cursor.execute("""
CREATE TABLE mental_health_data_backup AS
SELECT *
FROM mental_health_data;
""")

oltp_cursor.execute("""
DELETE FROM mental_health_data
WHERE country_code IS NULL OR country_code = 'OWID_WRL';
""")

oltp_connection.commit()
```

c. population data

Removed data aobut regions and only kept countries' information

In [41]:

```
oltp_cursor.execute("""
CREATE TABLE population_data_backup AS SELECT * FROM population_data;
""")

oltp_cursor.execute("""
DELETE FROM population_data WHERE TRIM("Population") IN (
    'Advanced economies',
    'ASEAN-5',
    'Africa (Region)',
    'Asia and Pacific',
    'Australia and New Zealand',
    'Caribbean',
    'Central America',
    'Central Asia and the Caucasus',
```

```
    'East Asia',
    'Eastern Europe',
    'Emerging and Developing Asia',
    'Emerging and Developing Europe',
    'Emerging market and developing economies',
    'Euro area',
    'Europe',
    'European Union',
    'Kosovo',
    'Latin America and the Caribbean',
    'Major advanced economies (G7)',
    'Middle East and Central Asia',
    'Middle East (Region)',
    'North Africa',
    'North America',
    'Other advanced economies',
    'Pacific Islands',
    'South America',
    'South Asia',
    'Southeast Asia',
    'Sub-Saharan Africa',
    'Sub-Saharan Africa (Region)',
    'West Bank and Gaza',
    'Western Europe',
    'Western Hemisphere (Region)',
    'World'
);
""")
```

Some names had trailing whitespaces. So, used TRIM function to avoid such issues during name matching

In [42]:

```
oltp_cursor.execute("""
UPDATE population_data
SET "Population" = CASE
    WHEN TRIM("Population") = 'Bahamas, The' THEN 'Bahamas'
    WHEN TRIM("Population") = 'Bolivia' THEN 'Bolivia (Plurinational State of)'
    WHEN TRIM("Population") = 'China, People''s Republic of' THEN 'China'
    WHEN TRIM("Population") = 'Congo, Dem. Rep. of the' THEN 'Democratic Republic of the
Congo'
    WHEN TRIM("Population") = 'Congo, Republic of' THEN 'Congo'
    WHEN TRIM("Population") = 'Côte d''Ivoire' THEN 'Côte d’Ivoire'
    WHEN TRIM("Population") = 'Czech Republic' THEN 'Czechia'
    WHEN TRIM("Population") = 'Gambia, The' THEN 'Gambia'
    WHEN TRIM("Population") = 'Hong Kong SAR' THEN 'China, Hong Kong Special Administrati
ve Region'
    WHEN TRIM("Population") = 'Iran' THEN 'Iran (Islamic Republic of)'
    WHEN TRIM("Population") = 'Korea, Republic of' THEN 'Republic of Korea'
    WHEN TRIM("Population") = 'Kyrgyz Republic' THEN 'Kyrgyzstan'
    WHEN TRIM("Population") = 'Lao P.D.R.' THEN 'Lao People''s Democratic Republic'
    WHEN TRIM("Population") = 'Macao SAR' THEN 'China, Macao Special Administrative Regio
n'
    WHEN TRIM("Population") = 'Micronesia, Fed. States of' THEN 'Micronesia (Federated St
ates of)'
    WHEN TRIM("Population") = 'Moldova' THEN 'Republic of Moldova'
    WHEN TRIM("Population") = 'North Macedonia' THEN 'North Macedonia'
    WHEN TRIM("Population") = 'São Tomé and Príncipe' THEN 'Sao Tome and Principe'
    WHEN TRIM("Population") = 'Slovak Republic' THEN 'Slovakia'
    WHEN TRIM("Population") = 'South Sudan, Republic of' THEN 'South Sudan'
    WHEN TRIM("Population") = 'Syria' THEN 'Syrian Arab Republic'
    WHEN TRIM("Population") = 'Taiwan Province of China' THEN 'Taiwan, Province of China'
    WHEN TRIM("Population") = 'Tanzania' THEN 'United Republic of Tanzania'
    WHEN TRIM("Population") = 'Türkiye, Republic of' THEN 'Turkey'
    WHEN TRIM("Population") = 'United Kingdom' THEN 'United Kingdom of Great Britain and
Northern Ireland'
    WHEN TRIM("Population") = 'United States' THEN 'United States of America'
    WHEN TRIM("Population") = 'Venezuela' THEN 'Venezuela (Bolivarian Republic of)'
    WHEN TRIM("Population") = 'Vietnam' THEN 'Viet Nam'
    ELSE "Population"
END;
```

```
""")

oltp_connection.commit()
```

d. economic data

In [43]:

```
oltp_cursor.execute("""
-- Creating a backup of the economic_data table
CREATE TABLE economic_data_backup AS
SELECT *
FROM economic_data;
""")

oltp_cursor.execute("""
-- Delete the row with country code 'XKX'
DELETE FROM economic_data
WHERE country_code = 'XKX';
""")

oltp_connection.commit()
```

6. DimGame

Used country code instead of country name on Hosts information which can be helpful during fact table creation and also during cube creation

In [44]:

```
oltp_cursor.execute("""
SELECT
    o.game_slug,
    o.game_name,
    o.game_season,
    o.game_year,
    c.country_code
FROM
    olympic_hosts o
JOIN
    countries c ON c.country_name = o.game_location
WHERE
    c.country_code IS NOT NULL;
""")

dim_game_data = oltp_cursor.fetchall()
```

In [45]:

```
dim_game_data_columns = [desc[0] for desc in oltp_cursor.description]
dim_game_data_columns
```

Out[45]:

```
['game_slug', 'game_name', 'game_season', 'game_year', 'country_code']
```

In [46]:

```
dim_game_df =  pd.DataFrame(dim_game_data, columns = dim_game_data_columns)

dim_game_df.head()
```

Out[46]:

	game_slug	game_name	game_season	game_year	country_code
0	beijing-2022	Beijing 2022	Winter	2022	CHN
1	tokyo-2020	Tokyo 2020	Summer	2020	JPN
2	pyeongchang-2018	PyeongChang 2018	Winter	2018	KOR

	game_slug	game_name	game_season	game_year	country_code
2	pyeongchang-2018	PyeongChang 2018	Winter	2018	KOR
3	rio-2016	Rio 2016	Summer	2016	BRA
4	sochi-2014	Sochi 2014	Winter	2014	RUS

In [47]:

```
dim_game_df.to_sql("dimgame", con=olap_engine, if_exists="append", index=False)
```

Out[47]:

```
53
```

Intermediary Table For Fact Tables

Selecting certain columns from the original table to populate fact table. I decided to create intermediary table to assist in the creation of fact tables. Since most of the things related to the olympic medals are in olympic_medals table, i selected the only thing that might be interesting to us.

In [48]:

```
oltp_cursor.execute("""
SELECT
    discipline_title,
    slug_game,
    event_title,
    event_gender,
    medal_type,
    participant_type,
    participant_title,
    athlete_full_name,
    country_3_letter_code as country_code
FROM
    olympic_medals;
""")
```

In [49]:

```
olap_olympic_data = oltp_cursor.fetchall()
```

In [50]:

```
olap_olympic_data_columns = [desc[0] for desc in oltp_cursor.description]
olap_olympic_data_columns
```

Out[50]:

```
['discipline_title',
 'slug_game',
 'event_title',
 'event_gender',
 'medal_type',
 'participant_type',
 'participant_title',
 'athlete_full_name',
 'country_code']
```

In [51]:

```
olap_olympic_df =  pd.DataFrame(olap_olympic_data, columns = olap_olympic_data_columns)
```

In [52]:

```
olap_olympic_df.head()
```

Out[52]:

	discipline_title	slug_game	event_title	event_gender	medal_type	participant_type	participant_title	athlete_full_name	coun
		tokyo							

	discipline_title	slug_game	event_title	event_gender	medal_type	participant_type	participant_title	athlete_full_name	coun
0	Football	tokyo-2020	Women	Women	GOLD	GameTeam	Canada	None	
1	Archery	tokyo-2020	Women's Team	Women	SILVER	GameTeam	ROC	None	
2	Softball	beijing-2008	softball women	Women	GOLD	GameTeam	Japan team	None	
3	Football	seoul-1988	football men	Men	GOLD	GameTeam	Soviet Union team	None	
4	Equestrian Jumping	seoul-1988	team mixed	Open	BRONZE	GameTeam	France team	None	

In [53]:

```
olap_cursor.execute("""
CREATE TABLE olap_olympic_medals (
    discipline_title TEXT,
    slug_game TEXT,
    event_title TEXT,
    event_gender VARCHAR(250),
    medal_type TEXT,
    participant_type TEXT,
    participant_title TEXT,
    athlete_full_name TEXT,
    country_code CHAR(3)
);
""")
```

In [54]:

```
olap_connection.commit()
```

In [55]:

```
olap_olympic_df.to_sql("olap_olympic_medals", con=olap_engine, if_exists="append", index=
False)
```

Out[55]:

644

Adding participant_id to replace type and title

Step 1: Modify the Table Structure **First, alter `olap_olympic_medals` table to add the `participant_id` column.**

```
ALTER TABLE olap_olympic_medals
ADD COLUMN participant_id INTEGER;
```

Step 2: Update the `participant_id` Column **Use an `UPDATE` statement with a `JOIN` to populate the new `participant_id` based on `participant_title` and `participant_type`.**

```
UPDATE olap_olympic_medals
SET participant_id = p.participant_id
FROM DimParticipant p
WHERE olap_olympic_medals.participant_title = p.participant_title
AND olap_olympic_medals.participant_type = p.participant_type;
```

Step 3: Remove Old Columns **After successfully updating the `participant_id` column, remove the old columns (`participant_title` and `participant_type`)**

```
ALTER TABLE olap_olympic_medals
DROP COLUMN participant_title,
DROP COLUMN participant_type;
```

In [56]:

```
olap_cursor.execute("""
ALTER TABLE olap_olympic_medals
ADD COLUMN participant_id INTEGER;
""")

olap_cursor.execute("""
UPDATE olap_olympic_medals
SET participant_id = p.participant_id
FROM DimParticipant p
WHERE olap_olympic_medals.participant_title = p.participant_title
AND olap_olympic_medals.participant_type = p.participant_type;
""")

olap_cursor.execute("""
ALTER TABLE olap_olympic_medals
DROP COLUMN participant_title,
DROP COLUMN participant_type;
""")

olap_connection.commit()
```

Adding `athelte_id` to remove information about athlete in the original data

Step 1: Add a New Column for Athlete ID **First, alter `olap_olympic_medals` table to add the `athlete_id` column.**

```
ALTER TABLE olap_olympic_medals
ADD COLUMN athlete_id INTEGER;
```

Step 2: Populate the Athlete ID Column **Update the `athlete_id` in `olap_olympic_medals` by joining it with the `DimAthlete` table based on the `athlete_full_name`.**

```
UPDATE olap_olympic_medals o
SET athlete_id = a.athlete_id
FROM DimAthlete a
WHERE o.athlete_full_name = a.athlete_name;
```

In [57]:

```
olap_cursor.execute("""
ALTER TABLE olap_olympic_medals
ADD COLUMN athlete_id INTEGER;
""")

olap_cursor.execute("""
UPDATE olap_olympic_medals o
SET athlete_id = a.athlete_id
FROM DimAthlete a
WHERE o.athlete_full_name = a.athlete_name;
""")

olap_cursor.execute("""
ALTER TABLE olap_olympic_medals
DROP COLUMN athlete_full_name;
""")

olap_connection.commit()
```

Adding `event_id` to replace event information

Step 1: Add New Column for Event ID **Alter `olap_olympic_medals` table to add the `event_id` column.**

```
ALTER TABLE olap_olympic_medals
ADD COLUMN event_id INTEGER;
```

```
ADD COLUMN event_id INTEGER;
```

Step 2: Populate the Event ID Column update the `event_id` in `olap_olympic_medals` by performing a join with the `DimEvent` table. The join condition will match `event_title`, `event_discipline`, and `event_gender` between the two tables.

```
UPDATE olap_olympic_medals o
SET event_id = e.event_id
FROM DimEvent e
WHERE o.event_title = e.event_title
AND o.event_discipline = e.event_discipline
AND o.event_gender = e.event_gender;
```

Step 3: Remove Old Columns Once the `event_id` is populated, remove the `event_title`, `event_discipline`, and `event_gender` columns from the `olap_olympic_medals` table.

```
ALTER TABLE olap_olympic_medals
DROP COLUMN event_title,
DROP COLUMN event_discipline,
DROP COLUMN event_gender;
```

In [58]:

```
olap_cursor.execute("""
ALTER TABLE olap_olympic_medals
ADD COLUMN event_id INTEGER;
""")

olap_cursor.execute("""
UPDATE olap_olympic_medals o
SET event_id = e.event_id
FROM DimEvent e
WHERE o.event_title = e.event_title
AND o.discipline_title = e.event_discipline
AND o.event_gender = e.event_gender;
""")

olap_cursor.execute("""
ALTER TABLE olap_olympic_medals
DROP COLUMN event_title,
DROP COLUMN discipline_title,
DROP COLUMN event_gender;
""")

olap_connection.commit()
```

Adding year column so that it can help in querying later on

```
-- Adding a new column for year
ALTER TABLE olap_olympic_medals
ADD COLUMN year INTEGER;

-- Updating the year column based on the game_slug match in DimGame
UPDATE olap_olympic_medals o
SET year = g.game_year
FROM DimGame g
WHERE o.game_slug = g.game_slug;
```

In [59]:

```
olap_cursor.execute("""
ALTER TABLE olap_olympic_medals
ADD COLUMN year INTEGER;
""")
```

In [60]:

```
olap_cursor.execute("""
UPDATE olap_olympic_medals o
SET year = g.game_year
FROM DimGame g
WHERE o.slug_game = g.game_slug;
""")
```

Adding number of medals won by the country in different year

```
SELECT
    country_code,
    year AS yr,
    COUNT(CASE WHEN medal_type = 'BRONZE' THEN 1 END) AS total_bronze_medals,
    COUNT(CASE WHEN medal_type = 'SILVER' THEN 1 END) AS total_silver_medals,
    COUNT(CASE WHEN medal_type = 'GOLD' THEN 1 END) AS total_gold_medals,
    COUNT(*) AS total_medals
FROM
    olap_olympic_medals
GROUP BY
    country_code,
    yr
ORDER BY
    yr, total_medals DESC;
```

In [61]:

```
olap_cursor.execute("""
SELECT
    country_code,
 participant_id,
 athlete_id,
    event_id,
 slug_game as game_slug,
    year,
    COUNT(CASE WHEN medal_type = 'BRONZE' THEN 1 END) AS total_bronze_medals,
    COUNT(CASE WHEN medal_type = 'SILVER' THEN 1 END) AS total_silver_medals,
    COUNT(CASE WHEN medal_type = 'GOLD' THEN 1 END) AS total_gold_medals,
    COUNT(*) AS total_medals
FROM
    olap_olympic_medals
GROUP BY
    country_code,
 participant_id,
 athlete_id,
    event_id,
 slug_game,
    year
ORDER BY
    year, total_medals DESC;
""")

fact_olympic_measure_data = olap_cursor.fetchall()
```

In [62]:

```
fact_olympic_measure_data_columns = [desc[0] for desc in olap_cursor.description]
fact_olympic_measure_data_columns
```

Out[62]:

```
['country_code',
 'participant_id',
 'athlete_id',
 'event_id',
```

```
 'event_id',
 'game_slug',
 'year',
 'total_bronze_medals',
 'total_silver_medals',
 'total_gold_medals',
 'total_medals']
```

In [63]:

```
fact_olympic_measure_df =  pd.DataFrame(fact_olympic_measure_data, columns = fact_olympic_measure_data_columns)
fact_olympic_measure_df
```

Out[63]:

	country_code	participant_id	athlete_id	event_id	game_slug	year	total_bronze_medals	total_silver_medals	total_golc
0	GRC	NaN	NaN	611	athens-1896	1896	2	0	
1	GRC	NaN	NaN	455	athens-1896	1896	0	1	
2	USA	NaN	NaN	694	athens-1896	1896	0	2	
3	USA	NaN	NaN	873	athens-1896	1896	1	1	
4	GRC	NaN	NaN	129	athens-1896	1896	0	1	
...	...	...	...	...	...	...	...	...	
21578	SWE	NaN	7065.0	1571	beijing-2022	2022	0	1	
21579	NOR	268.0	NaN	1156	beijing-2022	2022	1	0	
21580	NOR	268.0	NaN	1151	beijing-2022	2022	0	0	
21581	TWN	NaN	4928.0	975	beijing-2022	2022	0	1	
21582	SWE	416.0	7065.0	1582	beijing-2022	2022	0	1	

21583 rows × 10 columns

In [64]:

```
fact_olympic_measure_df.to_sql("factolympicmedalsmeasures", con=olap_engine, if_exists="append", index=False)
```

Out[64]:

583

In [65]:

```
### Economic Measure
```

In [ ]: