THE UNIVERSITY OF WESTERN AUSTRALIA

CITS5504: Project Report on DataWarehouse

Submitted By:
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.

Step 2: Clients and their business queries

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

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.

- **Potential Questions:**

a. **Medals vs. Hosting Years**: How did the USA's medal tally vary in the Olympic Games immediately before, during, and after the years they hosted the Olympics?
b. **Sport Discipline Success in Host Years**: Which sports disciplines yielded the most medals for the USA in the years they hosted the Olympics?
c. **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)?
d. **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.
e. **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?

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.

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

c) **Discipline-Specific Health Analysis:** Are certain sports disciplines more positively correlated with the national health status in terms of medal success?
d) **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?
e) **Comparative Health Analysis:** How does Australia's Olympic success compare with that of other countries with similar health metrics?

Step 3: Data Warehouse Design and Implementation

By employing the four-stage approach of dimensional modelling, as outlined by Kimball, I created a design for a data warehouse based on the provided datasets.

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 Olympic event for which a medal was awarded. This would include the specific games, year, event, discipline, and the participating athletes/countries. Based on the data and client specifications provided, I developed a starnet query model. Although the hierarchical structure for some dimensions was clear, dimensions like Economy, Health, and Medals primarily consisted of indicators representing economic status, health status, and similar measures.

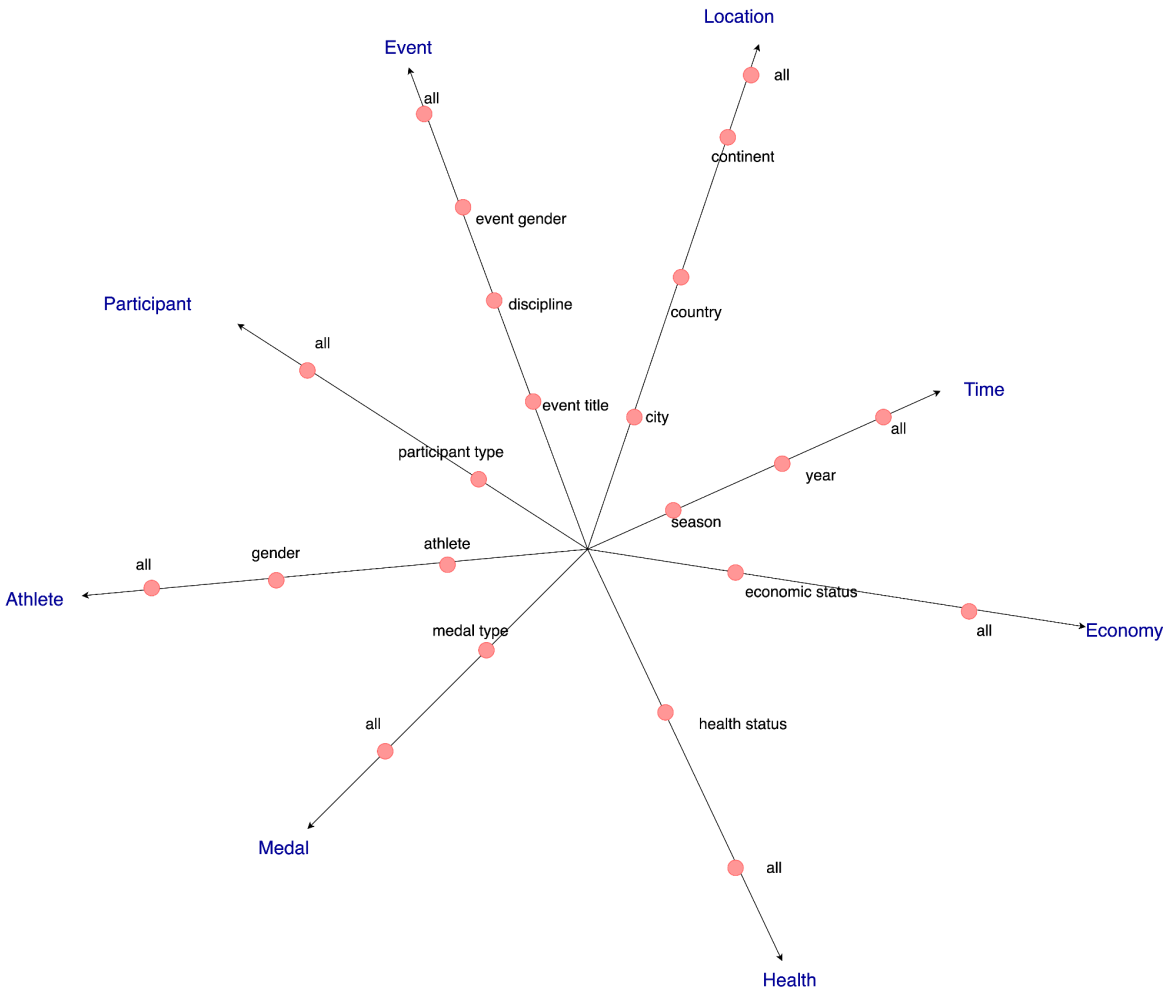


Figure 1. Starnet Model Diagram to answer overall business queries

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 decided to create the following fact tables which can contain these data.

1. FactOlympicMedalsMeasures
2. FactEconomicMeasure
3. FactHealthMeasure

Step 4: Data Profiling and Cleaning

Considering the level of detail required and the datasets available, the following are typical data-cleansing actions that could be applied to 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).

I have preemptively cleansed the data to minimize the need for cleaning during the ETL (Extract, Transform, Load) process. The specifics for each dataset are documented in the **notebook**, with the economic data serving as an example.

1. 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:

- **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.
- **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 began by substituting placeholders such as ".." with NaN, adjusting data types, and managing missing values. I executed a comparable script across all datasets to profile them. This produces an output that allows me to view the extent of missing data, which is particularly useful in large datasets.

```
# 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
```

Table 1.

Table 1. Loading economic data from CSV file

```
<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)
```

Table 2.

Table 2. Output of the above script

I applied the aforementioned techniques to clean the data.

```
# 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
```

Table 3.

Table 3. Cleaning economic data

Upon replacing ".." with NaN, it became apparent that a considerable amount of data is missing, which was previously obscured by the placeholder "..".

```
<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)
```

Table 4.

Table 4. Output after cleaning

I followed up by substituting the absent values with the median.

```
# Calculate the 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
```

Table 5.

Table 5. Replacing empty values with median values

Following this step, there were no remaining instances of missing 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
```

Table 6.

Table 6. Checking the empty data.

Next, I established a connection to the PostgreSQL database and loaded the cleaned data into the OLTP (Online Transaction Processing) system.

```
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()
```

Table 7.

Table 7. Script to create a database table for loading data

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)
```

Table 8.

Table 8. Loading economic data to OLTP

Additionally, I stored the cleaned data in a CSV file for potential future access.

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

Table 9.

Table 9. Script to save data into CSV format

2. Global Population

To address missing data in the Global Population dataset, linear interpolation was employed due to the typically gradual linear changes observed in population trends.

```
# 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)
```

Table 10.

Table 10. Applying linear interpolation to fill missing values

Step 5: ETL (Extract, Transform, Load) Process

Despite cleaning the data during the loading process into OLTP, further cleaning was necessary to ensure it could be loaded into the appropriate dimension and fact tables. This required the implementation of the ETL process. Initially, I crafted a Logical Map to aid in the ETL process and the development of Dimension and Fact tables. You can access the map through the following link: Logical Map.

				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				

Figure 2. Logical Map for ETL process

Please consult the attached notebook for a comprehensive overview of the entire ETL process, as it contains more detailed information. The entire ETL process can be found in the appendix below.

There are four notebooks available:

1. "**oltp.ipynb**" for the OLTP process.
2. "**etl.ipynb**" for the ETL process.
3. "**cube.ipynb**" for the galaxy schema. Please note that this was not pursued further due to limited knowledge of creating cube with the galaxy schema.
4. "**combined_cube.ipynb**" where a single fact table has been created to simplify cube creation and querying.

Cleaning Countries Information.

This proved to be the most intricate aspect of the project, as various countries had different names and codes. To manage this, I developed an Excel sheet to maintain records of countries and utilized external data sources to establish standardized names and codes. Additionally, for countries with differing old and new codes, I manually compiled comparisons from online resources. The details of the country mappings can be found here: Mismatched Countries

Below is a sample table extracted from the Excel sheet, demonstrating the process of updating old country codes to new ones.

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	

Table 11. New country codes and old country codes

Step 6: Data Modelling

Initially, I formulated a galaxy schema for the fact table and dimension table using Atoti. The code generated the schema illustrated 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"])
```

Table 12. Atoti code to create Galaxy Schema

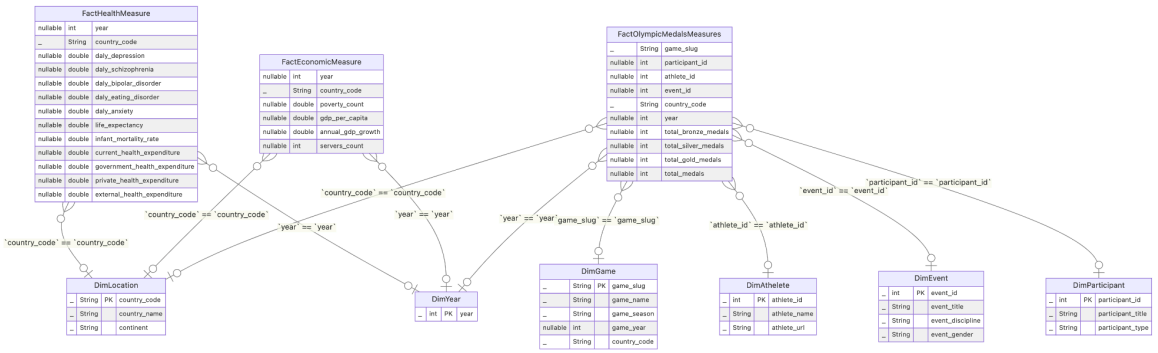


Figure 3. Galaxy Schema

However, given my limited understanding of Atoti and the lack of comprehensive documentation on their website regarding the creation of a cube from a galaxy schema, I chose to build a star schema instead. As part of this process, I merged the fact tables into a single fact table.

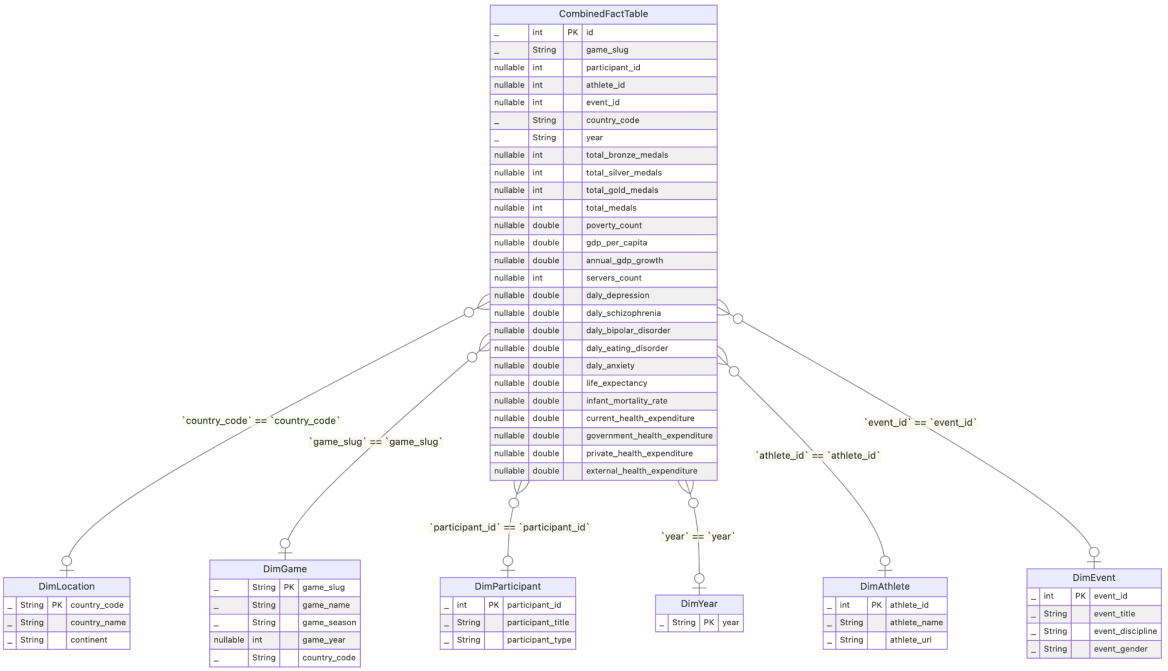


Figure 4. Star Schema

Step 7: Answering Business Queries

After creating the cube, I established the appropriate hierarchy and generated new measures. Utilizing the Atoti dashboard, I addressed all queries, leveraging its diverse filtering and graphing capabilities.

The following hierarchy was created as shown in the figure.

Hierarchies:

1. **DimAthlete**: athlete_name
2. **DimEvent**: event_title -> event_discipline -> event_gener
3. **DimGame**: game_name -> game_season
4. **DimLocation**: country_name -> continent
5. **DimParticipant**: participant_title -> participant_type
6. **DimYear**: year

```
[40]: hierarchies
```

```
[40]: ▼ Dimensions
        ▼ DimAthlete
          ▼ DimAthlete  []  1 item
              0 "athlete_name"
        ▼ DimEvent
          ▼ DimEvent  []  3 items
              0 "event_title"
              1 "event_discipline"
              2 "event_gender"
        ▼ DimGame
          ▼ DimGame  []  2 items
              0 "game_name"
              1 "game_season"
        ▼ DimLocation
          ▼ DimLocation  []  2 items
              0 "country_name"
              1 "continent"
        ▼ DimParticipant
          ▼ DimParticipant  []  2 items
              0 "participant_title"
              1 "participant_type"
        ▼ DimYear
          ▼ DimYear  []  1 item
              0 "year"
```

Figure 5. Hierarchies for the cube

I also loaded data into Tableau to create the following geographical visualisation which states the number of medals won by each country.

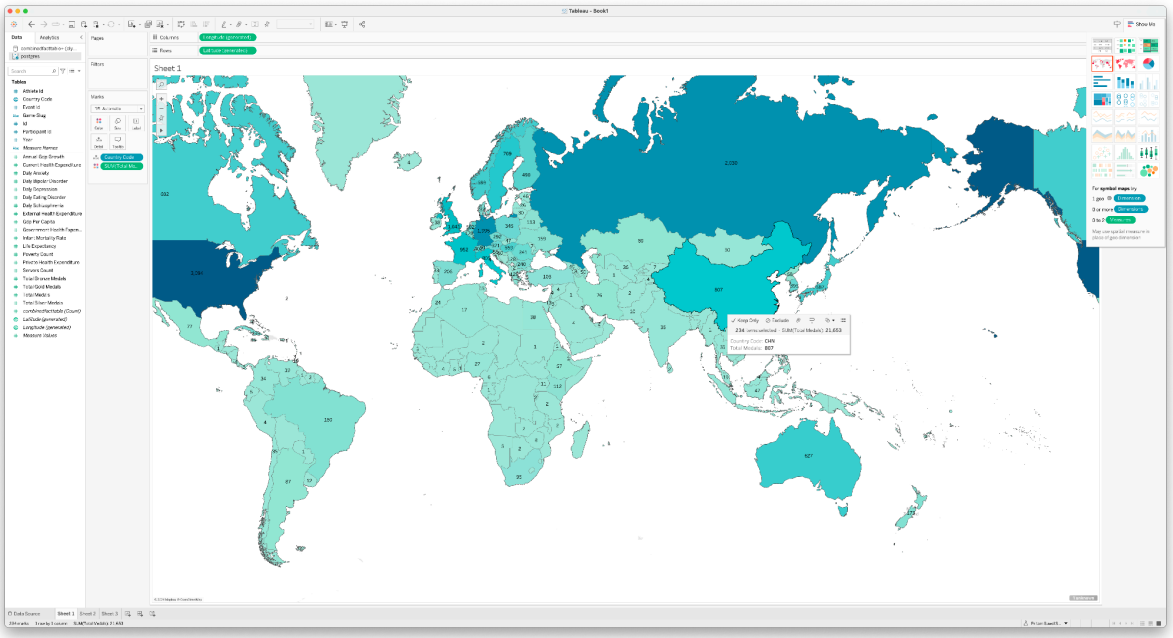


Figure 6. Olympic medals won by each country to date (Tableau)

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

1. **Medals vs. Hosting Years**: How did the USA's medal tally vary in the Olympic Games immediately before, during, and after the years they hosted the Olympics?
   This question can be effectively answered by analyzing the Olympic medals dataset in conjunction with the dataset detailing Olympic hosts. By examining medal performance over various years, including the hosting periods, we can directly observe any correlations between hosting the Games and medal achievements.

   The following starnet diagram defines the granularity needed to answer the above question

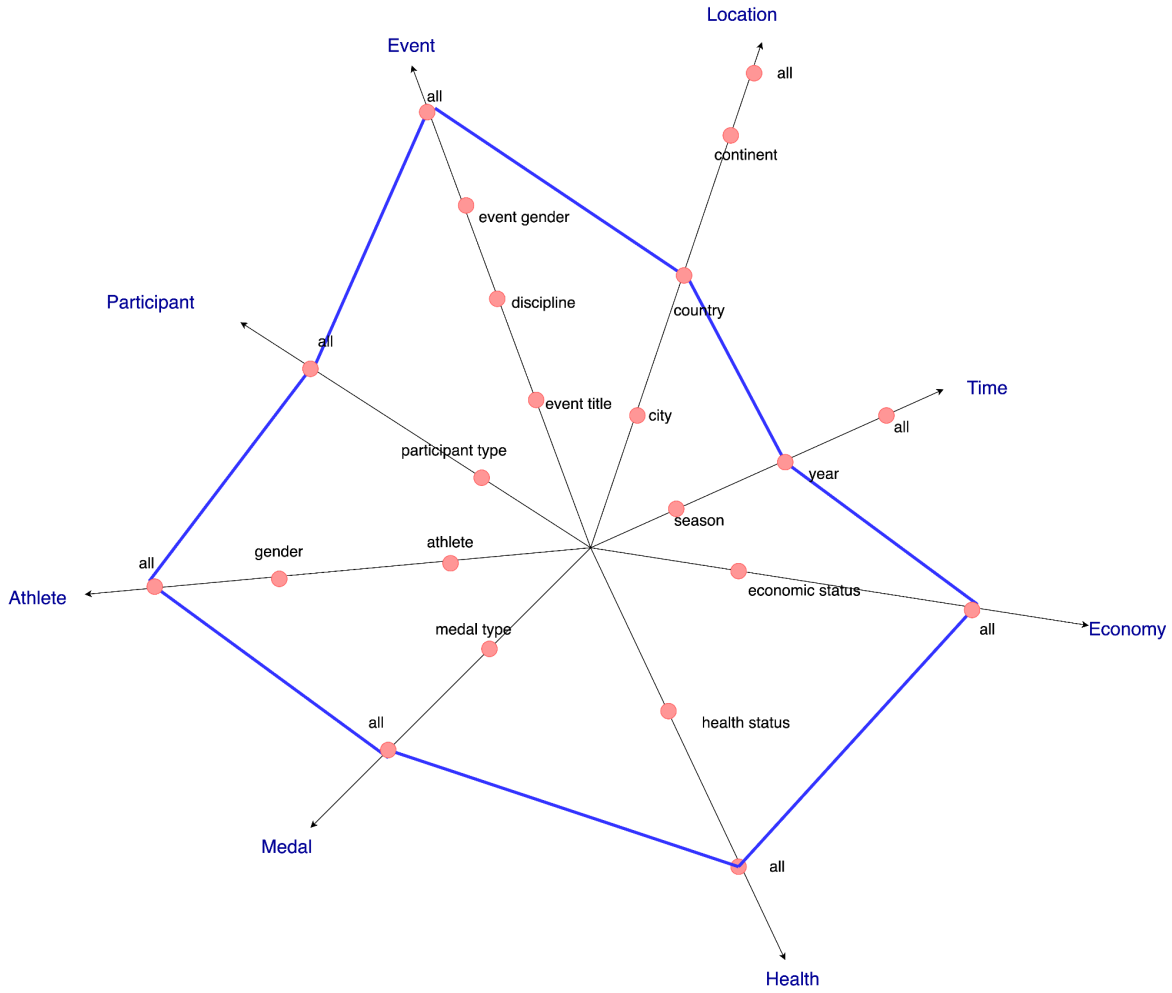


Figure 7. Medals vs Hosting Years - Starnet diagram

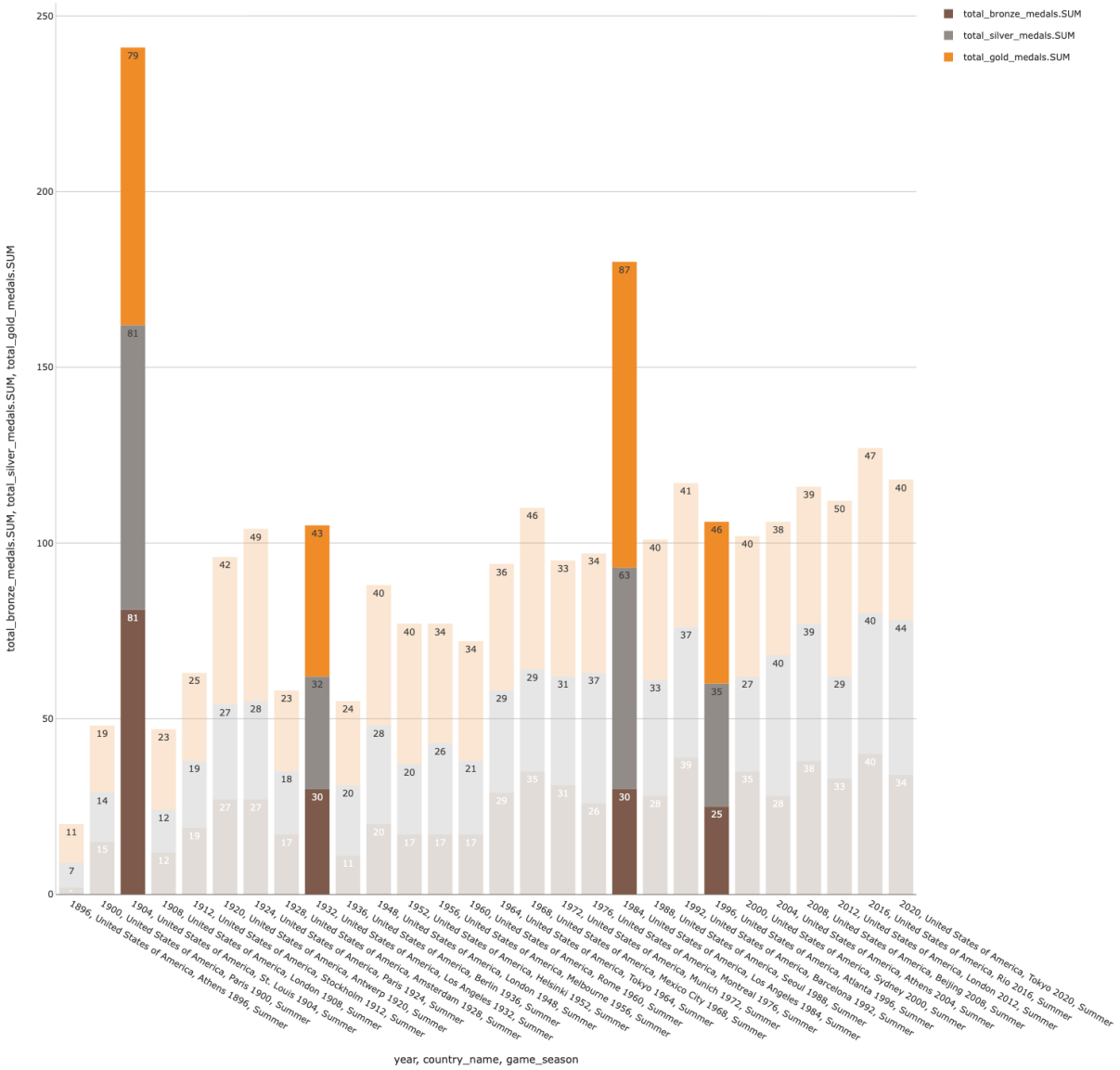


Figure 8. Olympic Medals in the Summer Games (USA)

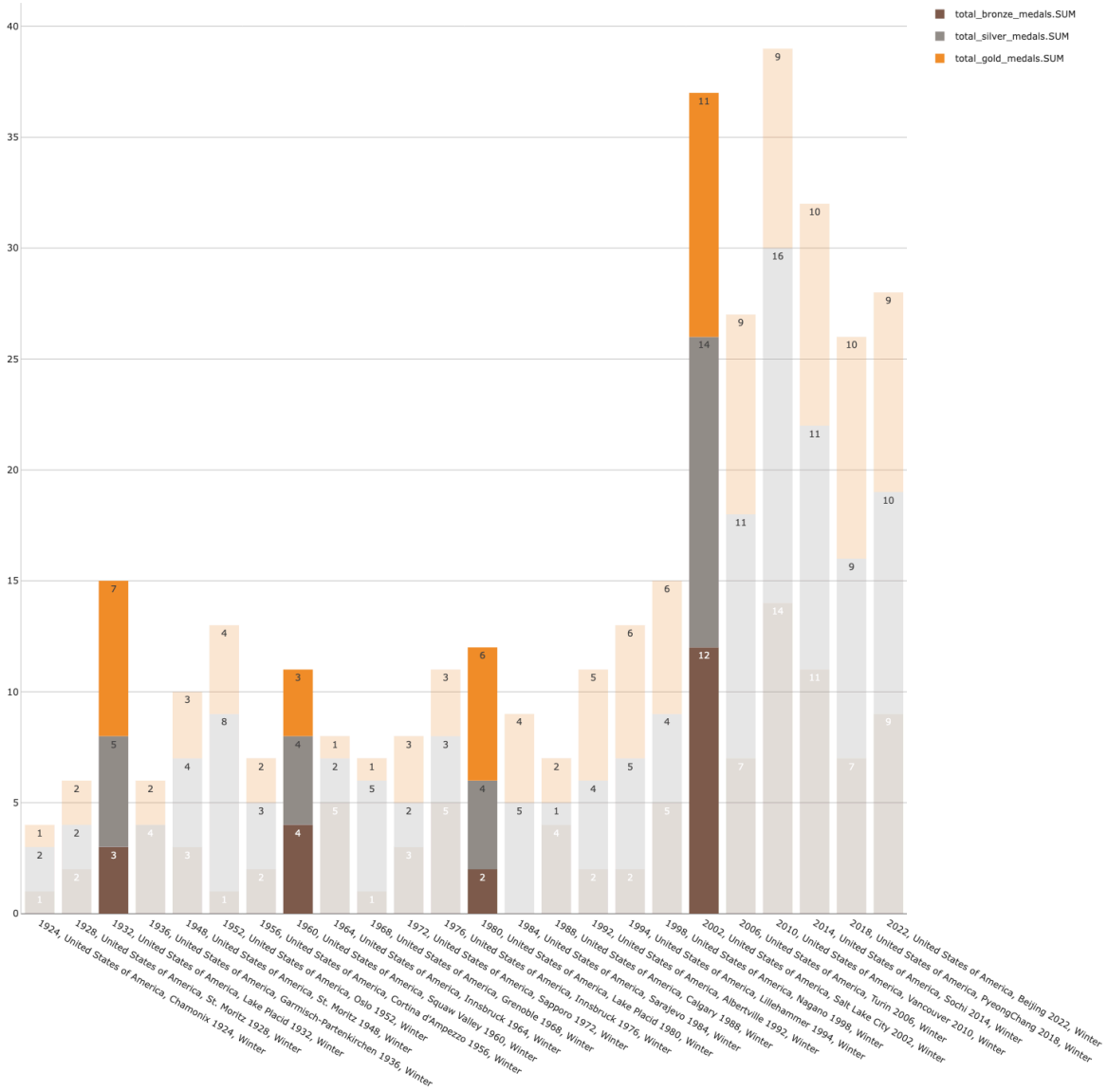


Figure 9. Olympic medals in Winter games (USA)

Using a stacked column graph in Atoti, we can effectively address the aforementioned query. The spikes in the graph indicate a clear correlation between hosting the Olympic Games and increased medal winnings. Hosting the Olympics typically provides a significant boost to the host country's performance, reflecting the combined effects of home advantage, heightened national focus on athletics, and enhanced investment in sports infrastructure and training programs.

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

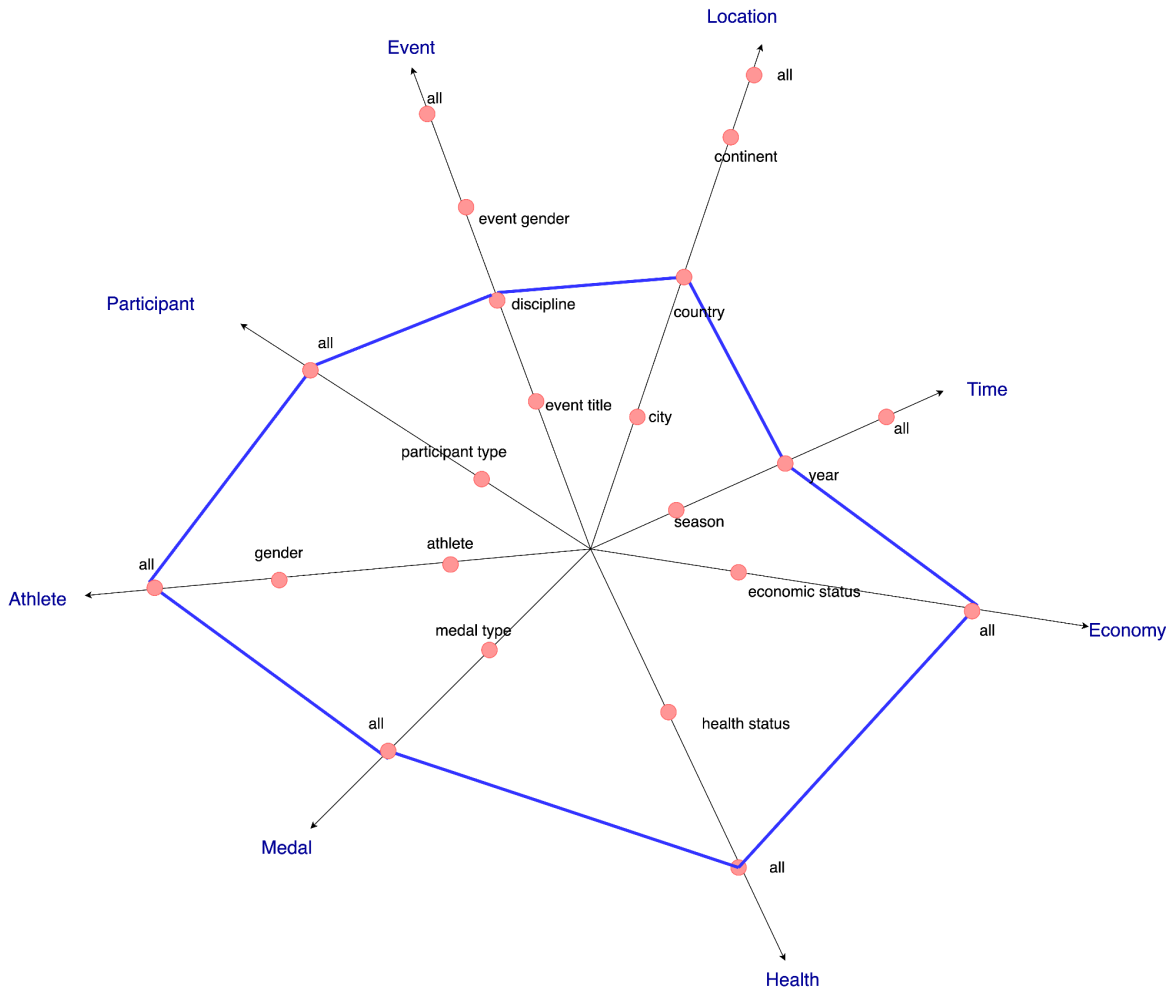


Figure 10. Starnet query model (Sport Discipline vs Hosting Years, USA)

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. By creating a stack column graph and pivot tables outlined below, we can analyse an exact number of medals won in different games when the USA hosted the Olympics.

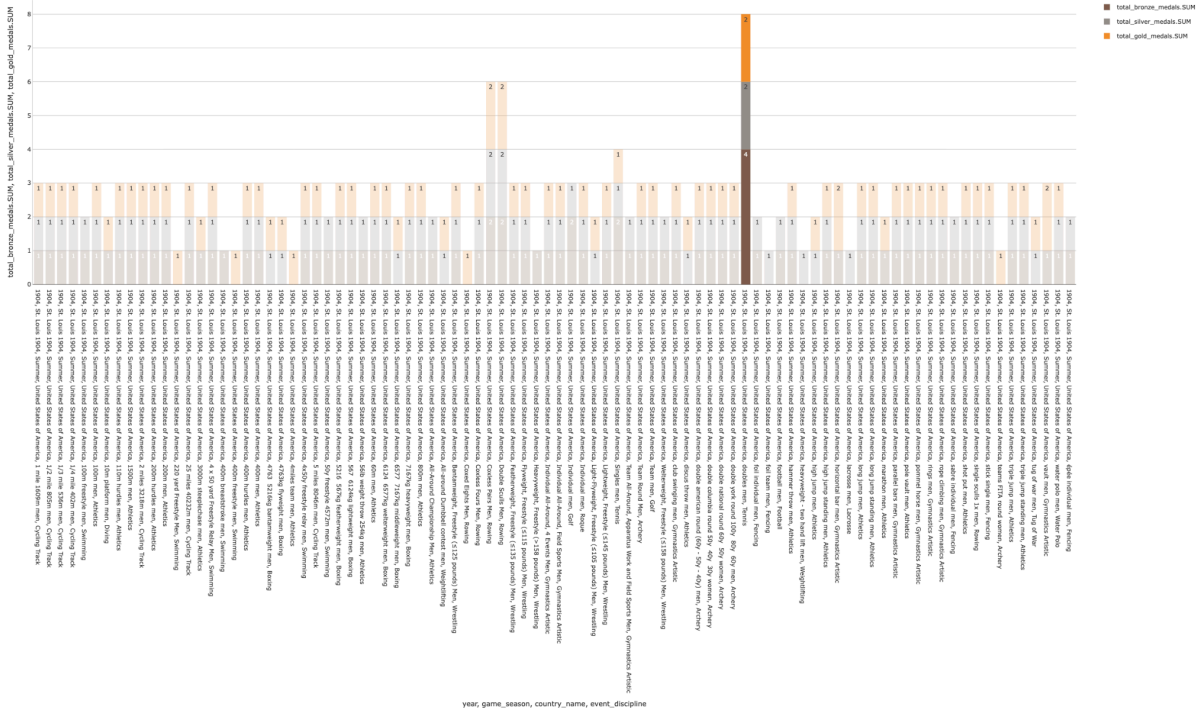


Figure 11. Highest number of medals in Tennis (doubles men) 1904 - Summer Games

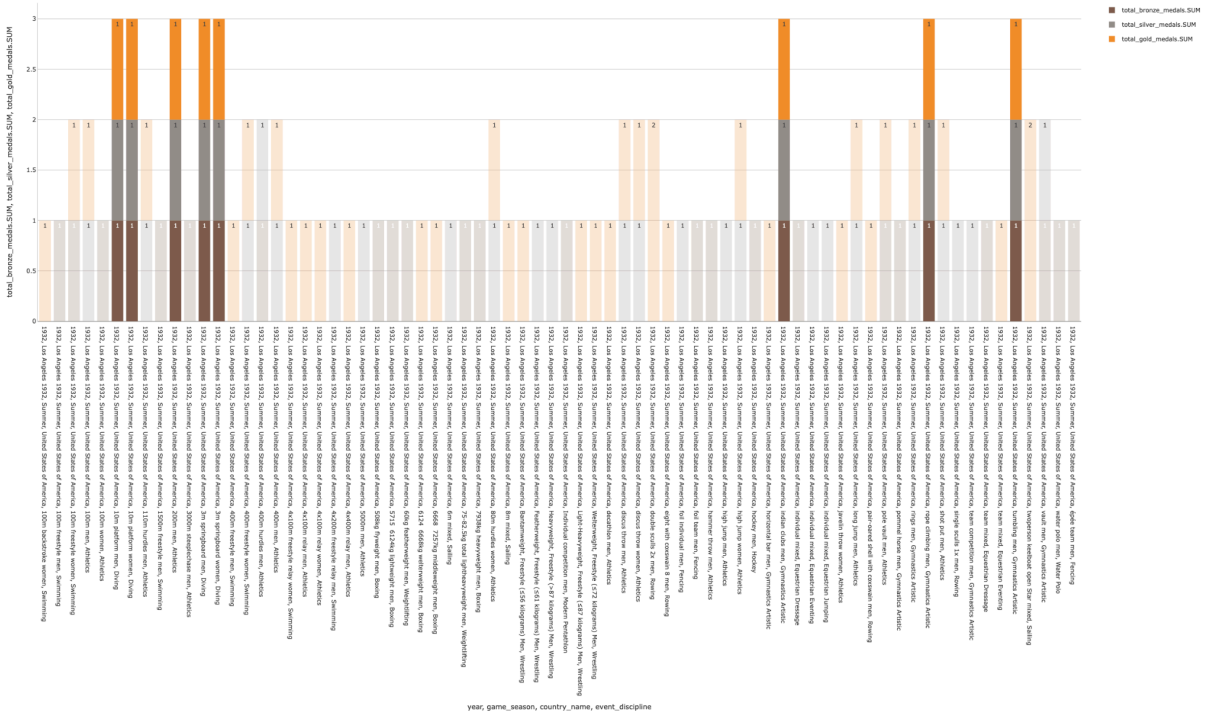


Figure 12. Number of medals in different disciplines in Summer Game - 1932

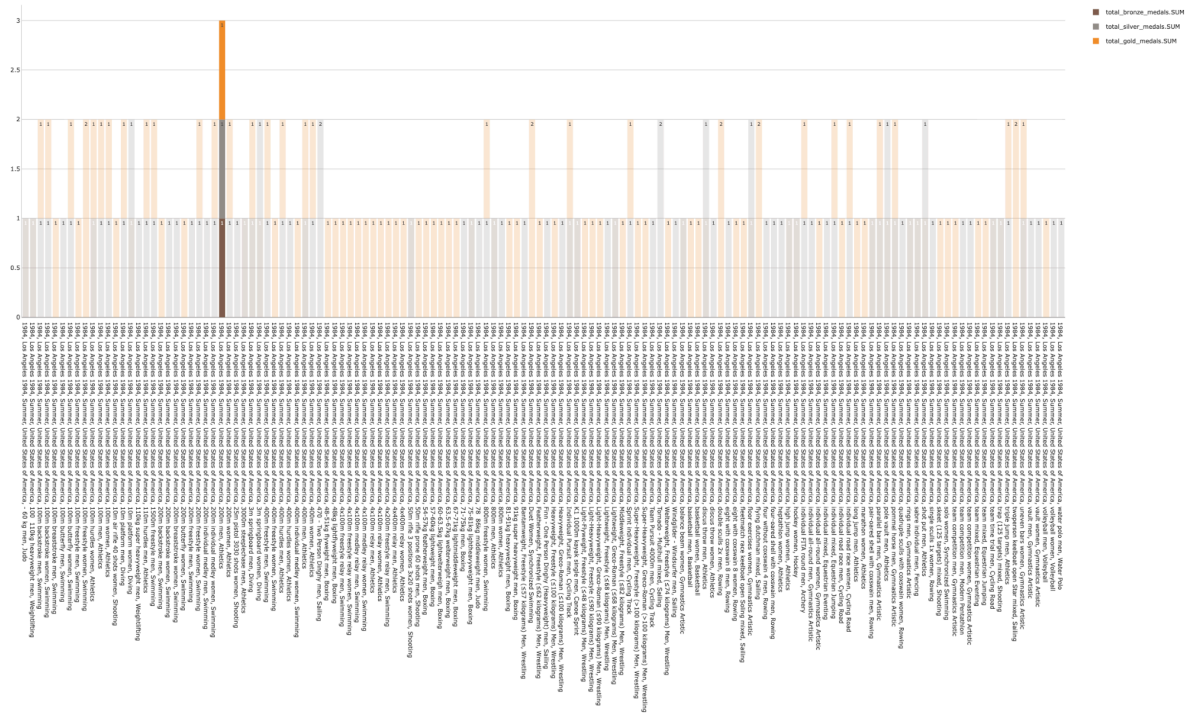


Figure 13. Highest Number of Medals in 200m men - Athletics (1984 - Summer Olympic)

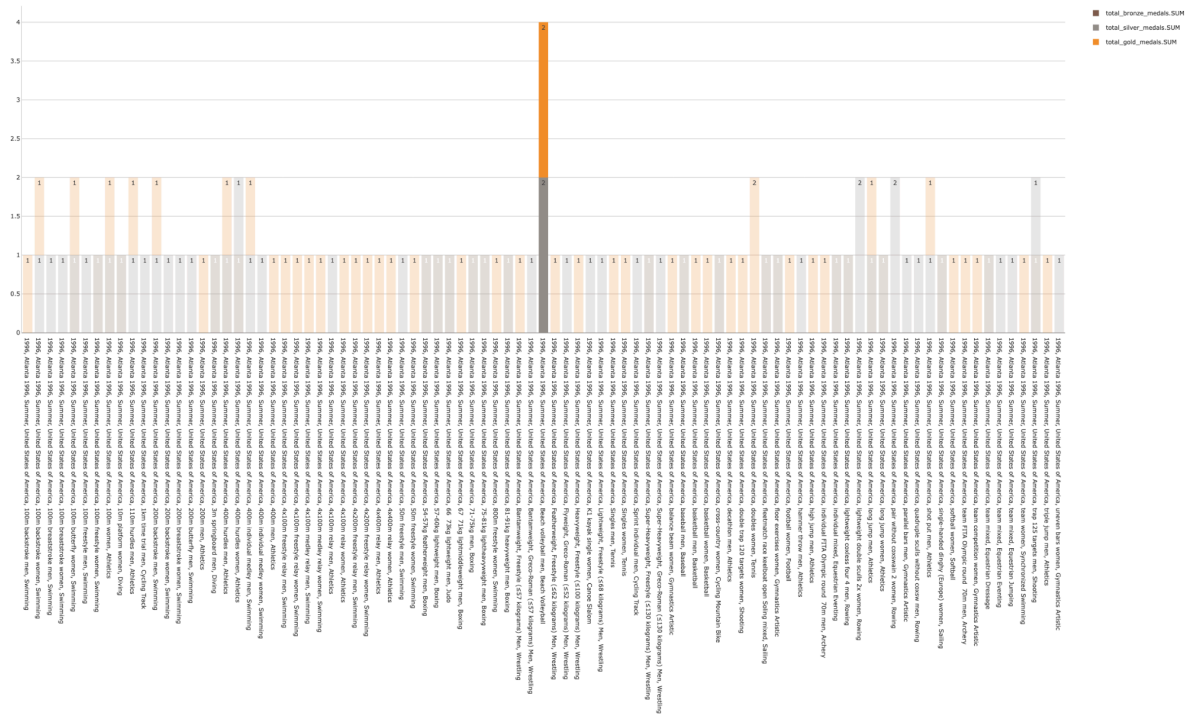


Figure 14. Highest Number of Medals in Beach Volleyball - Summer Olympic 1996

New pivot table

year	country_name	game_name	game_season	event_title	event_discipline	total_medals.SUM
1932	United States of America	Lake Placid 1932	Winter	10000m men	Speed skating	1
				1500m men	Speed skating	1
				5000m men	Speed skating	2
				500m men	Speed skating	1
				Individual women	Figure skating	1
				Pairs mixed	Figure skating	2
				four-man men	Bobsleigh	2
				ice hockey men	Ice Hockey	1
				two-man men	Bobsleigh	4
1960	United States of America	Squaw Valley 1960	Winter	500m men	Speed skating	1
				500m women	Speed skating	1
				Individual men	Figure skating	1
				Individual women	Figure skating	2
				Pairs mixed	Figure skating	2
				downhill women	Alpine Skiing	1
				giant slalom women	Alpine Skiing	1
				ice hockey men	Ice Hockey	1
				slalom women	Alpine Skiing	1
1980	United States of America	Lake Placid 1980	Winter	10000m men	Speed skating	1
				1000m men	Speed skating	1
				1000m women	Speed skating	1
				1500m men	Speed skating	1
				3000m women	Speed skating	1
				5000m men	Speed skating	1
				500m men	Speed skating	1
				500m women	Speed skating	1
				Individual men	Figure skating	1
				Individual women	Figure skating	1
				ice hockey men	Ice Hockey	1
				slalom men	Alpine Skiing	1
2002	United States of America	Salt Lake City 2002	Winter	1000m men	Short Track Speed Skating	1
					Speed skating	1
				1000m women	Speed skating	2
				1500m men	Short Track Speed Skating	1
					Speed skating	1
				1500m women	Speed skating	1
				5000m men	Speed skating	1
				500m men	Short Track Speed Skating	1
					Speed skating	2
				Aerials men	Freestyle Skiing	1
				Doubles mixed	Luge	4
				Giant parallel slalom men	Snowboard	1
				Half-pipe men	Snowboard	3
				Half-pipe women	Snowboard	1
				Individual men	Figure skating	1
					Skeleton	1
				Individual women	Figure skating	2
					Skeleton	2
				Moguls men	Freestyle Skiing	1
				Moguls women	Freestyle Skiing	1
				alpine combined men	Alpine Skiing	1
				four-man men	Bobsleigh	2
				giant slalom men	Alpine Skiing	1
				ice hockey men	Ice Hockey	1
				ice hockey women	Ice Hockey	1
				two-woman women	Bobsleigh	2

Figure 15. Number of medals in Winter Games hosted by USA

In analyzing the Atoti cube, we can identify which sports disciplines yielded the most medals for the USA during the years they hosted the Olympics. By utilizing stacked column graphs and pivot tables, we can precisely quantify the number of medals won in different games hosted by the USA. Examples include the highest number of medals in Tennis (doubles men) during the 1904 Summer Games, various disciplines during the 1932 Summer Games, 200m men - Athletics during the 1984 Summer Olympics, Beach Volleyball during the 1996 Summer Olympics, and medals won during Winter Games hosted by the USA. These insights provide valuable information on the sports that significantly contributed to the USA's medal tally during their hosting years.

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)?

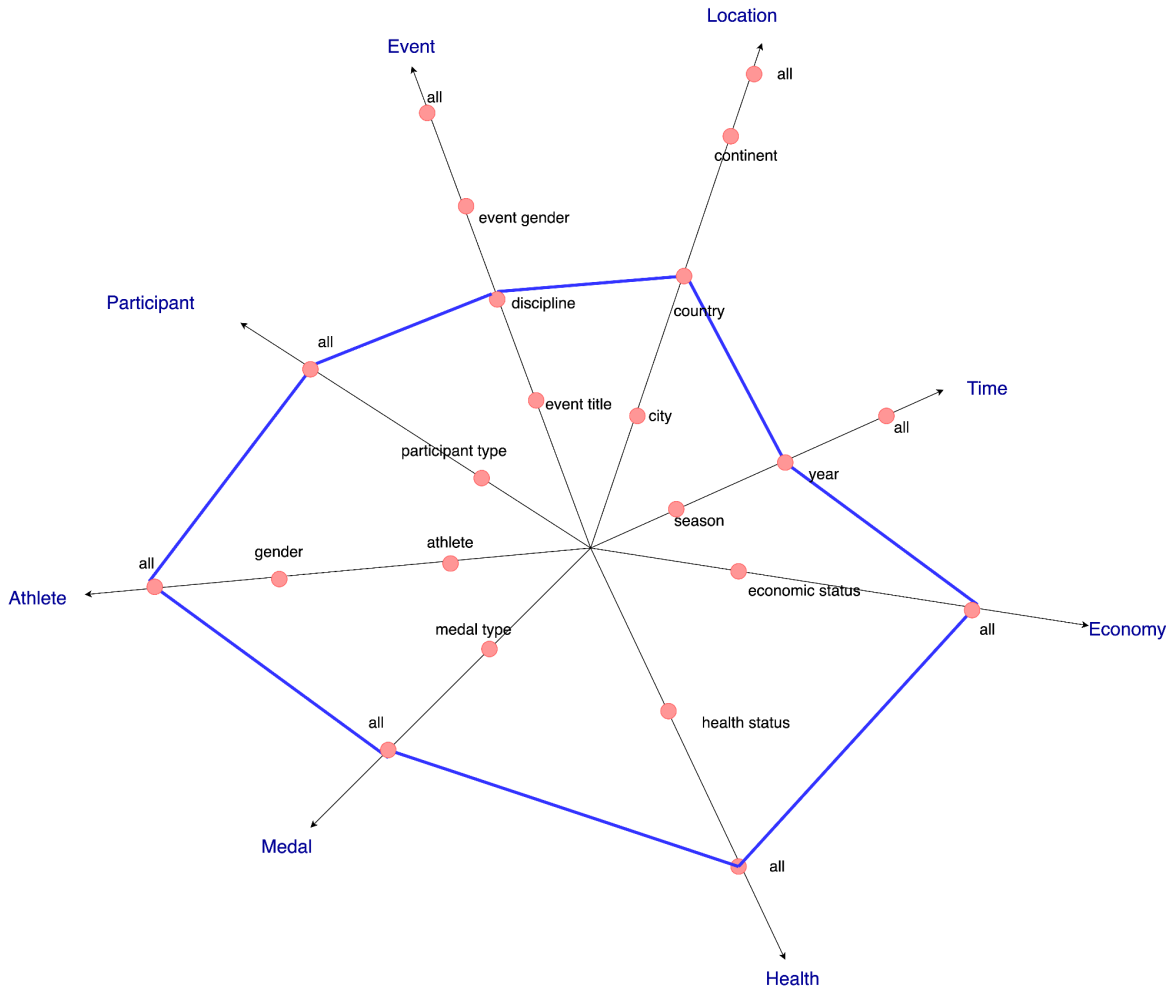


Figure 16. Startnet Query model (Economic Impact of Hosting) - USA

With the inclusion of economic data spanning the relevant years around each Olympic event hosted by the USA, this inquiry can explore into economic trends and potential impacts stemming from hosting the Games. Since the economic dataset only extends to 2020, we'll need to rely on external data sources to address this question comprehensively. This expanded data scope allows us to analyze the broader economic landscape surrounding the Olympic hosting years and assess any notable effects on the economy.

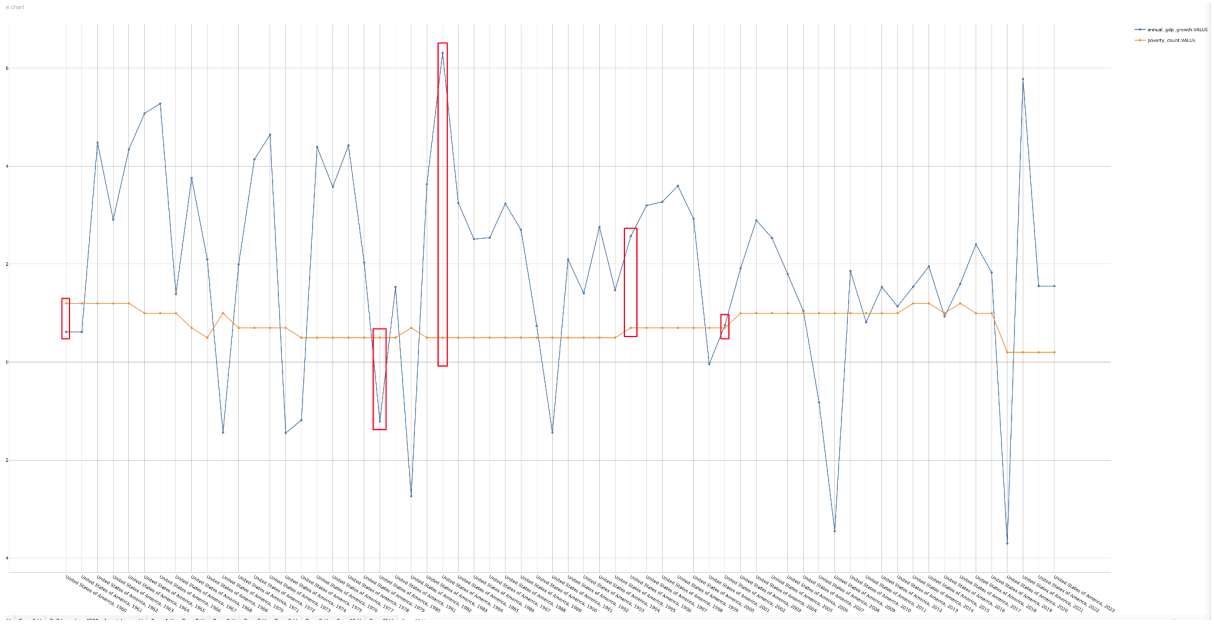


Figure 17. Poverty and annual GDP growth - USA (Red box highlights the year USA hosted the game)

There appears to be a noticeable spike or increasing trend in GDP following the USA's hosting of the Olympic Games. This could be attributed to the influx of tourism, investments in infrastructure, and heightened economic activity associated with hosting such a major international event.

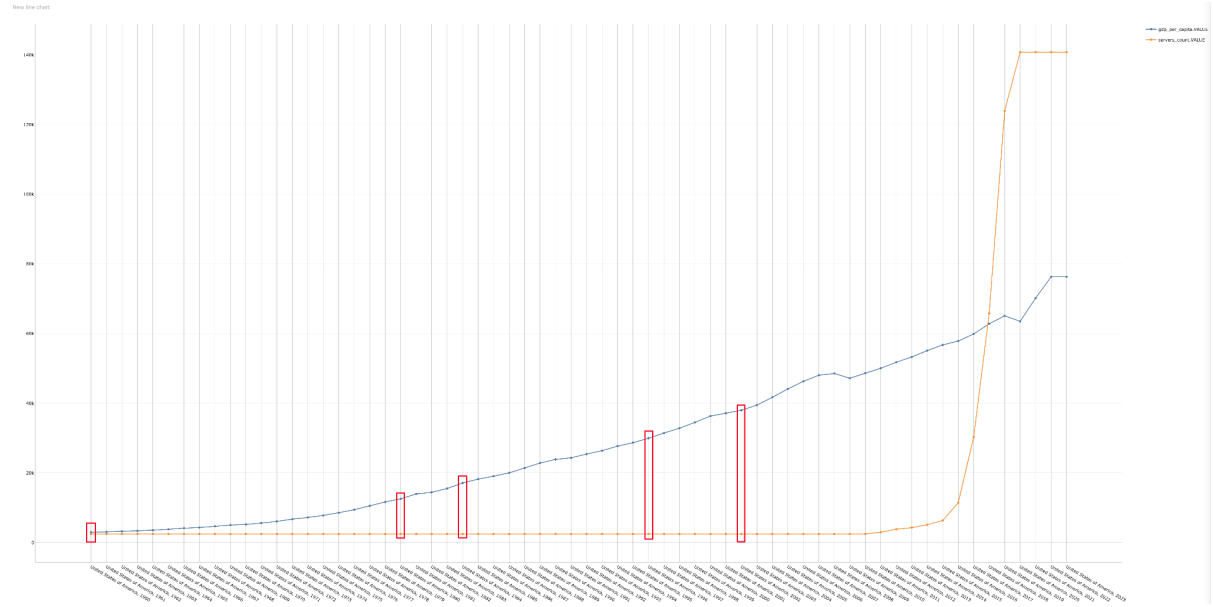


Figure 18. GDP per capita, Server Counts - USA

Despite whether the USA hosted the Olympic Games or not, there is an ongoing increasing trend in both GDP and server counts. Establishing a concrete causal relationship between

hosting the Games and these measures remains challenging due to various other factors influencing economic and technological trends.

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.

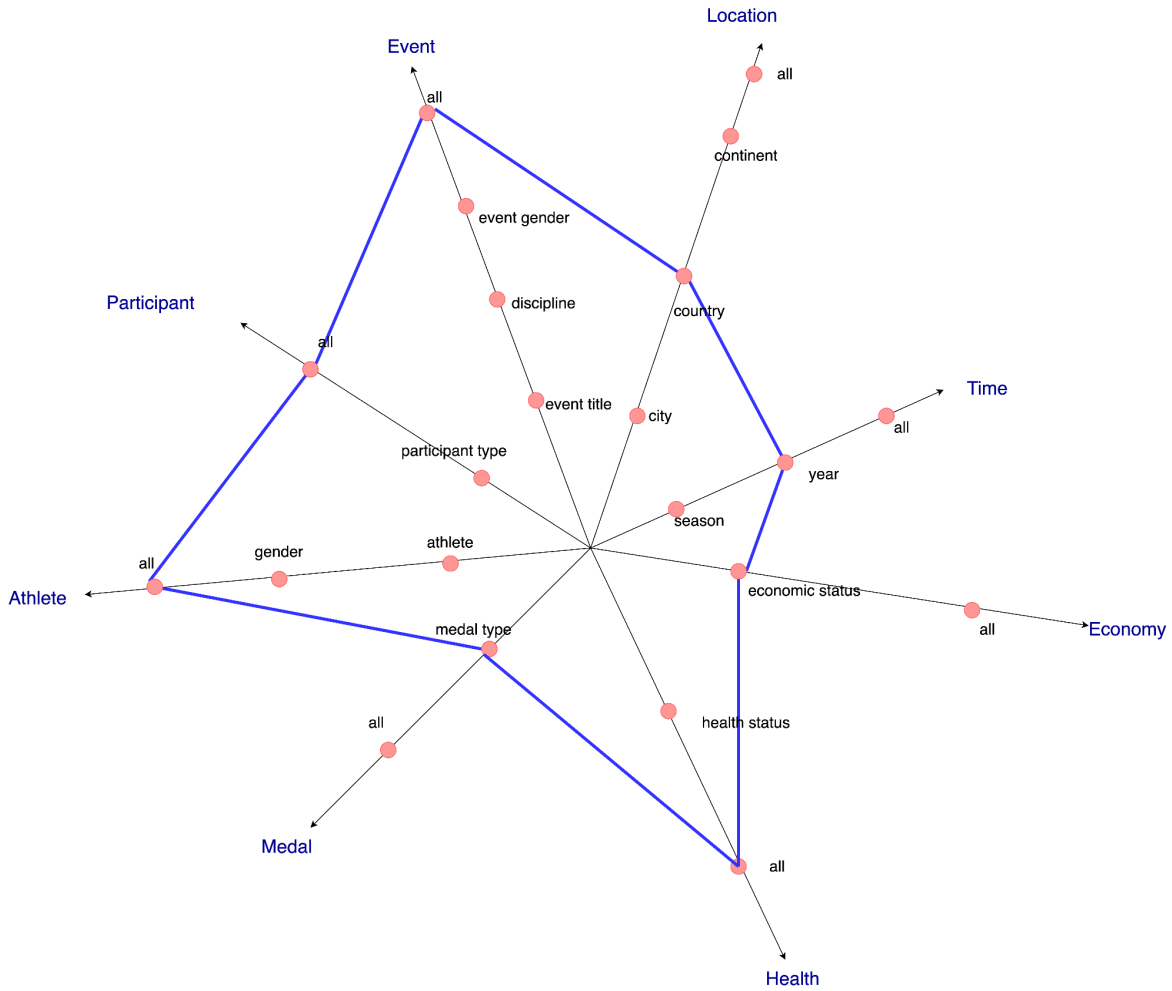


Figure 19. Starnet model (Economic Status and Medal Count) - USA

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.

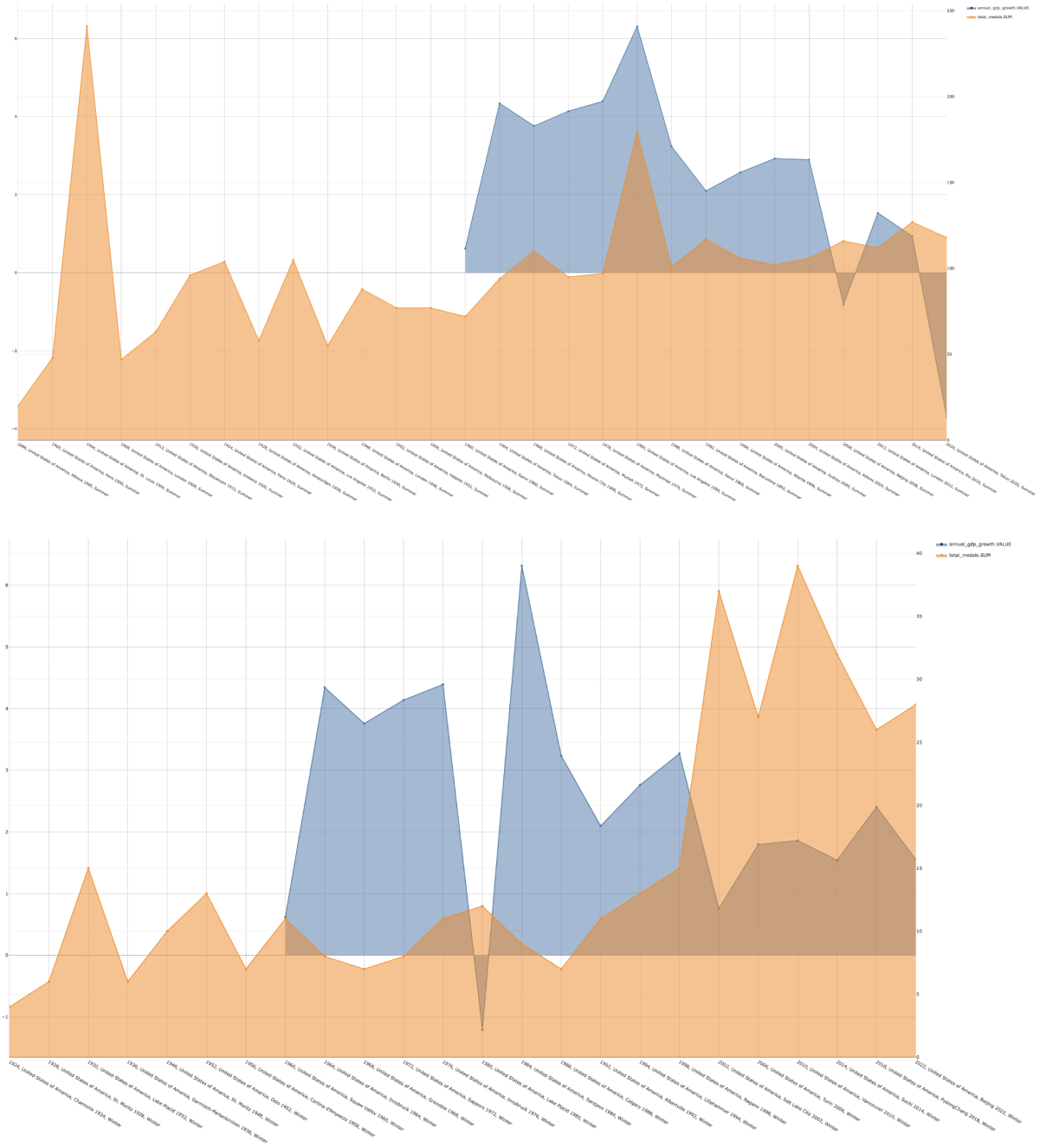


Figure 20. Annual GDP Growth - Summer and Winter Olympics USA

An evident spike in the annual growth rate aligns with years when the most Summer Olympic medals were won, subsequently declining over time. However, a definitive correlation is not apparent in the case of the Winter Games.

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?

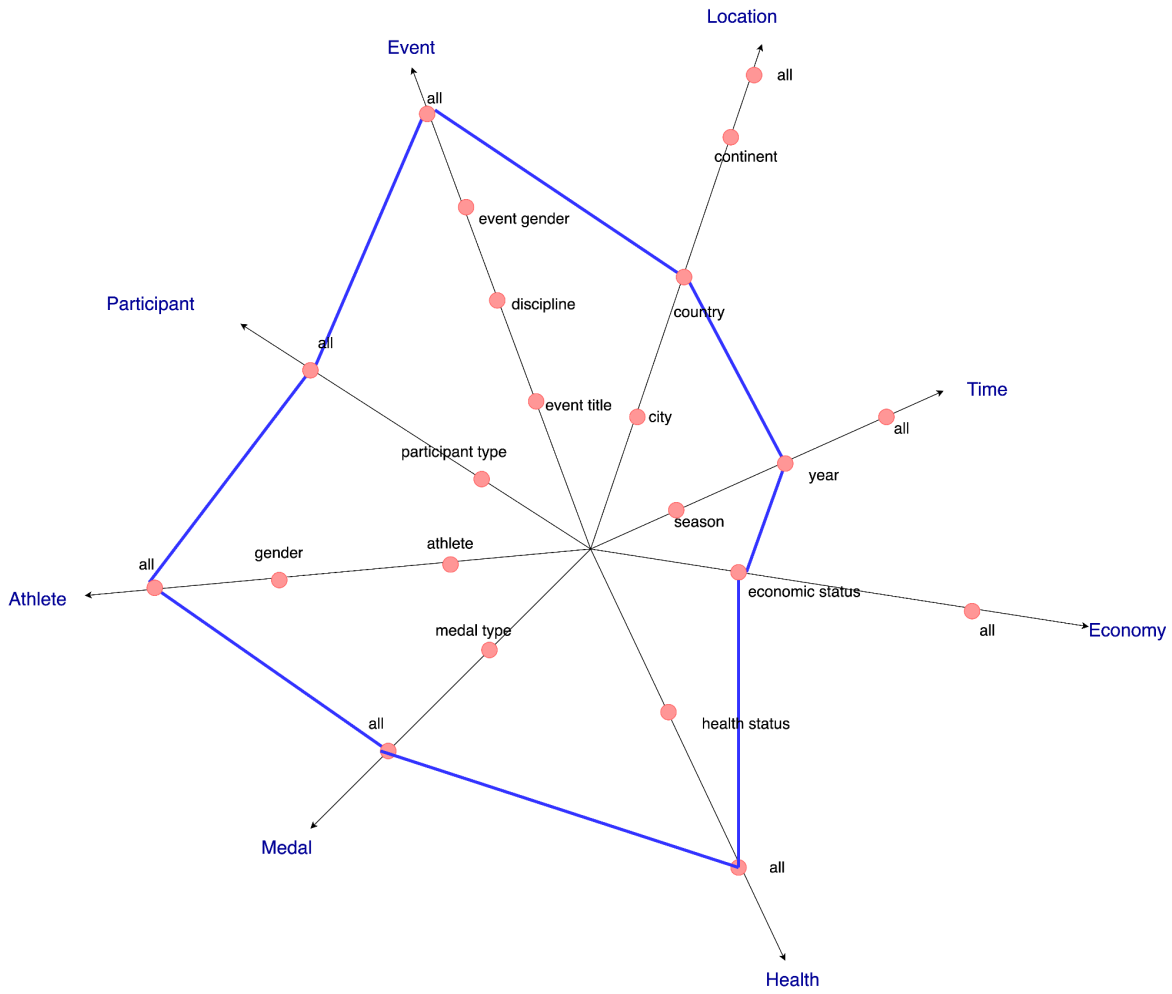


Figure 21. Impact of Technological Advancement - USA

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 which was provided with the external source.

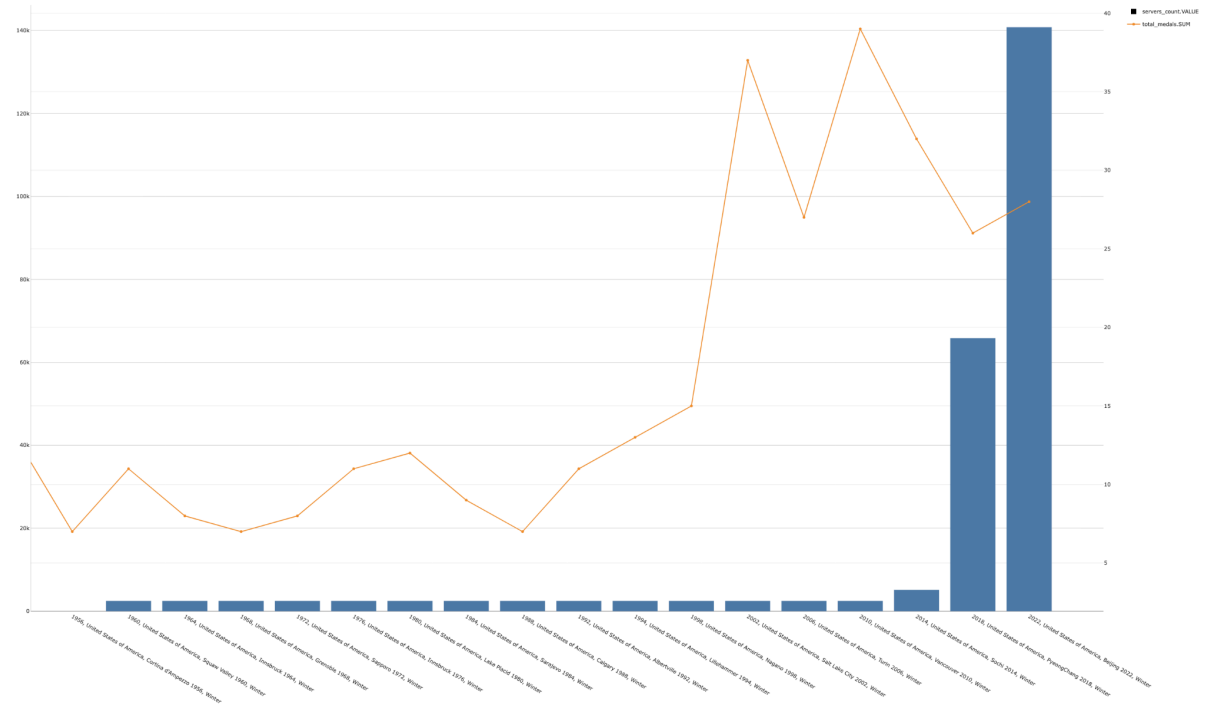


Figure 22. Server counts and Olympic Medals - USA

The data depicted in Figure reveals an increasing trend in both server counts and Olympic medals for the USA. This observation suggests a potential correlation between technological advancement, as indicated by an increase in secure internet servers per million people, and sports success. The rise in technological infrastructure might provide athletes with enhanced training resources, access to sports analytics, and support systems, thereby contributing to improved performance and increased medal counts over time.

Questions by the 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?

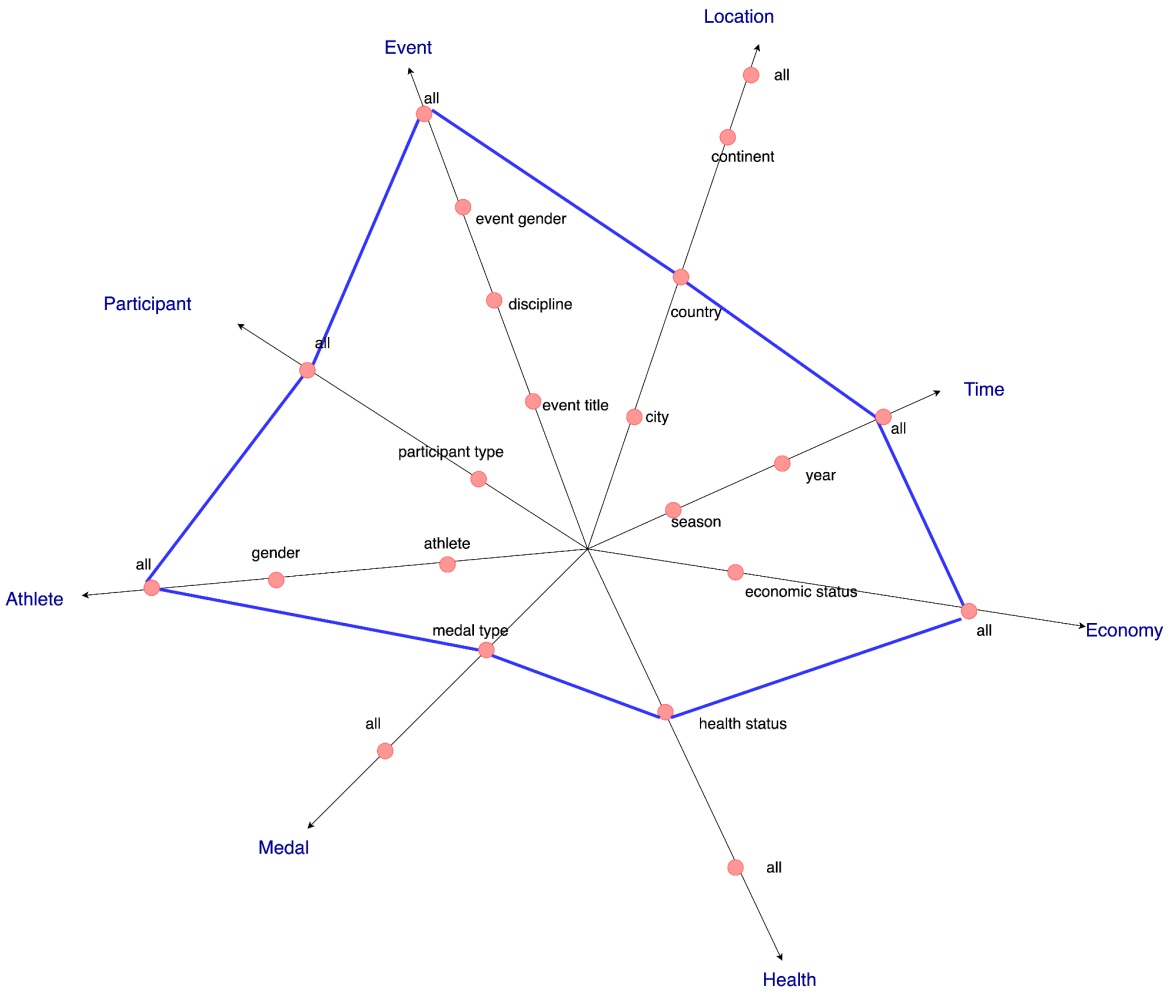


Figure 27. Starnet Query Model (Health and Performace Correlation) - Australia

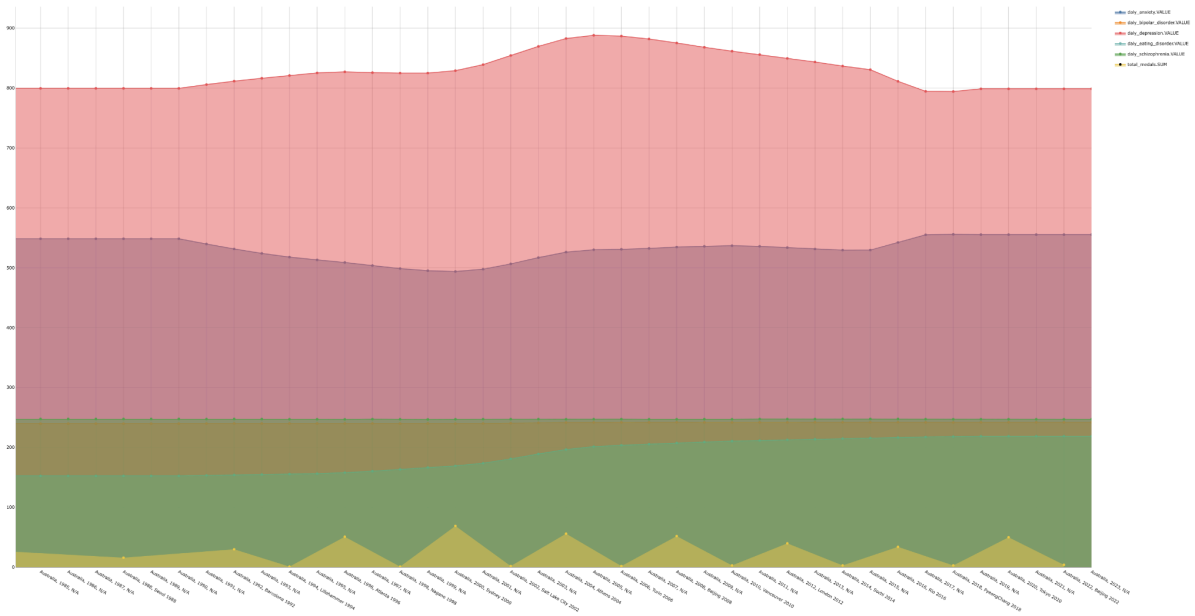


Figure 26. Mental Health Issues and Olympic medals - Australia

The data indicates that the highest number of medals was won during years when there were fewer reported mental health issues. The highlighted yellow years represent medal wins in both the Summer and Winter Olympics.

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?

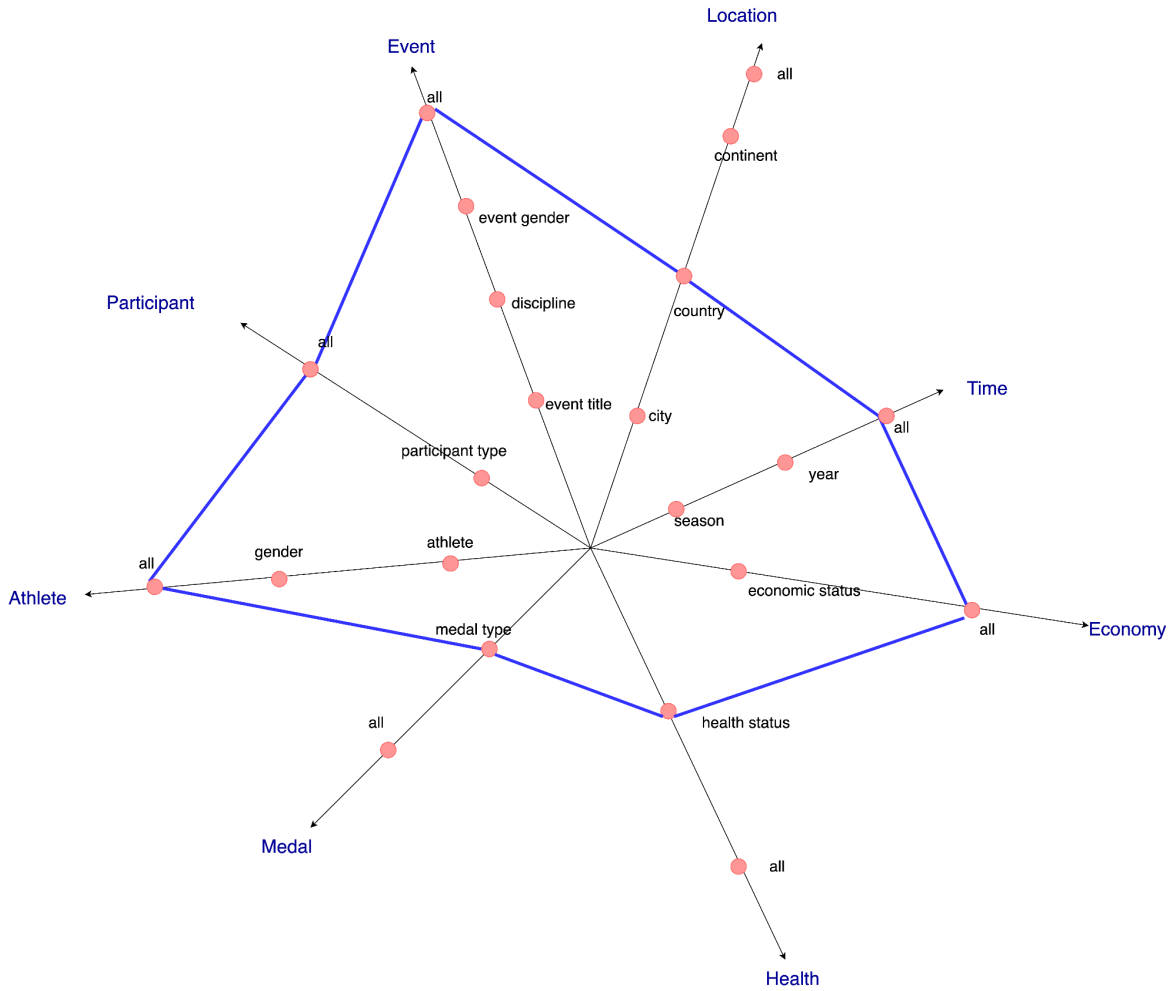


Figure 33. Hosting Influence on Health and Performance - Australia

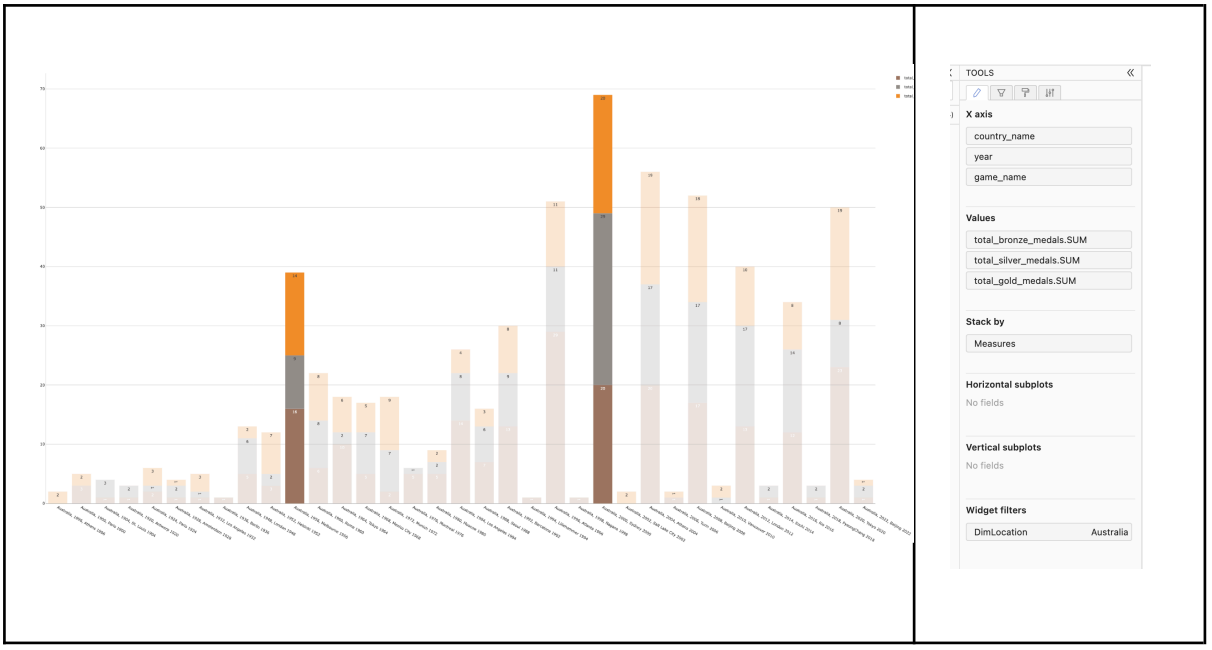


Figure 29. Olympic Medals when Australia hosted the game

There is a noticeable spike in the number of medals when Australia hosted the Olympic Games. The highest number of medals was achieved during Olympic events hosted by Australia. Looking ahead to the upcoming Games in Sydney, there is a strong likelihood of winning more awards based on past performance.

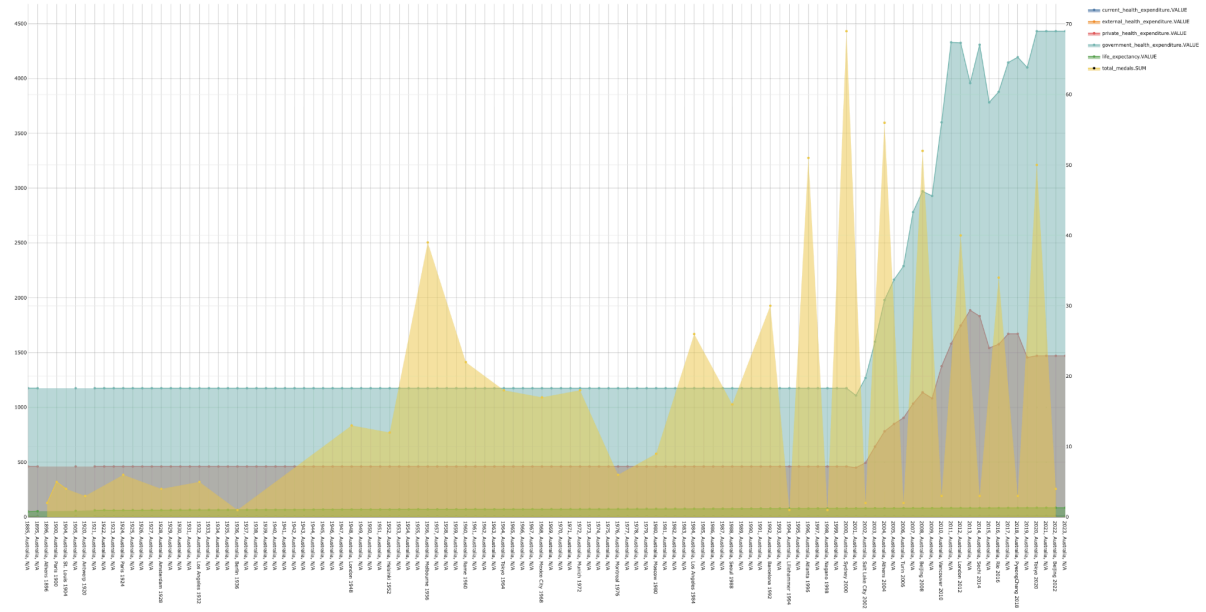


Figure 32. Health Expenditure and Olympic Medals - Australia

It's evident that government and private health expenditures accounts for the largest portion of health expenses, yet its impact on the total number of medals won appears to be minimal, contrary to expectations.

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

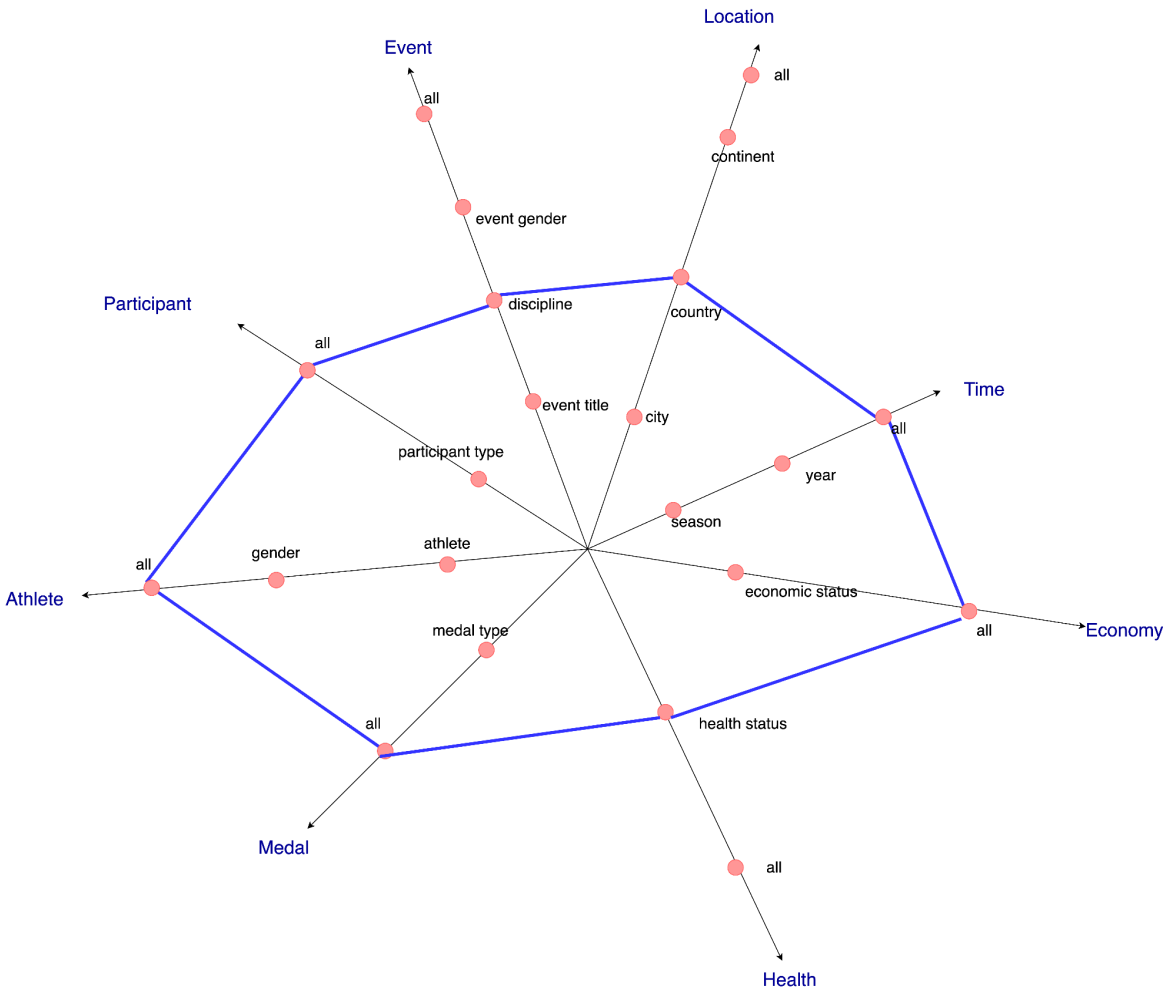


Figure 35. Starnet Query Model (Discipline-Specific Health Analysis)

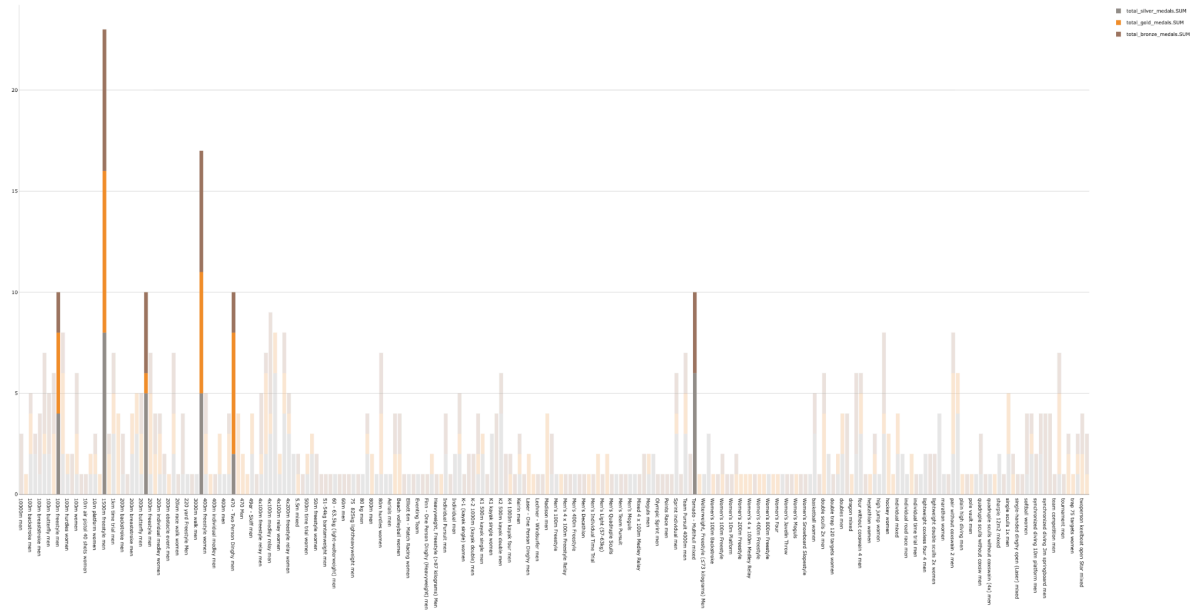


Figure 34. Number of Olympic medals and Event Discipline

Certain games consistently yield the highest number of medals as highlighted in the graph above.

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?

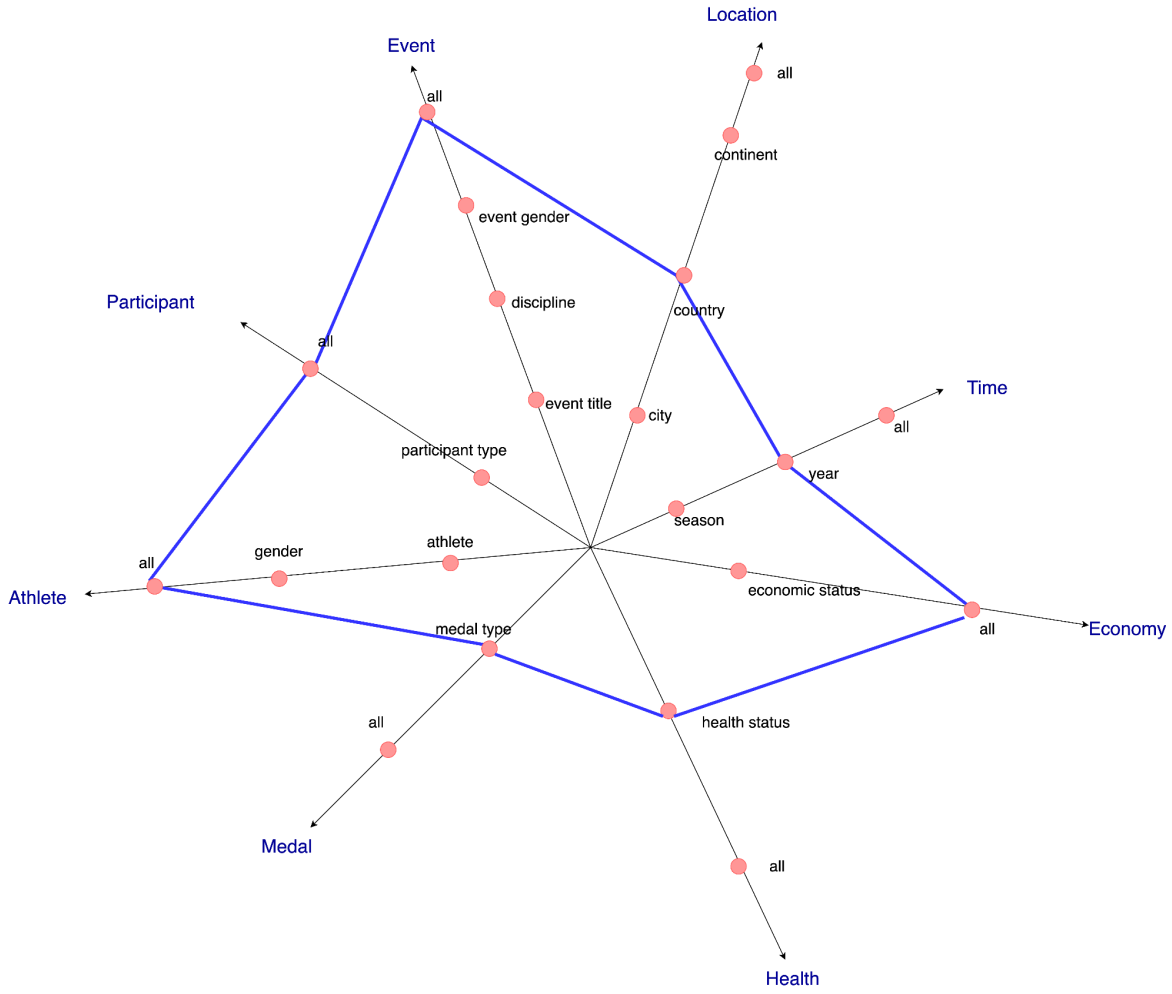


Figure 37. Starnet Query Model (Long-Term Health Trends vs. Olympic Success)

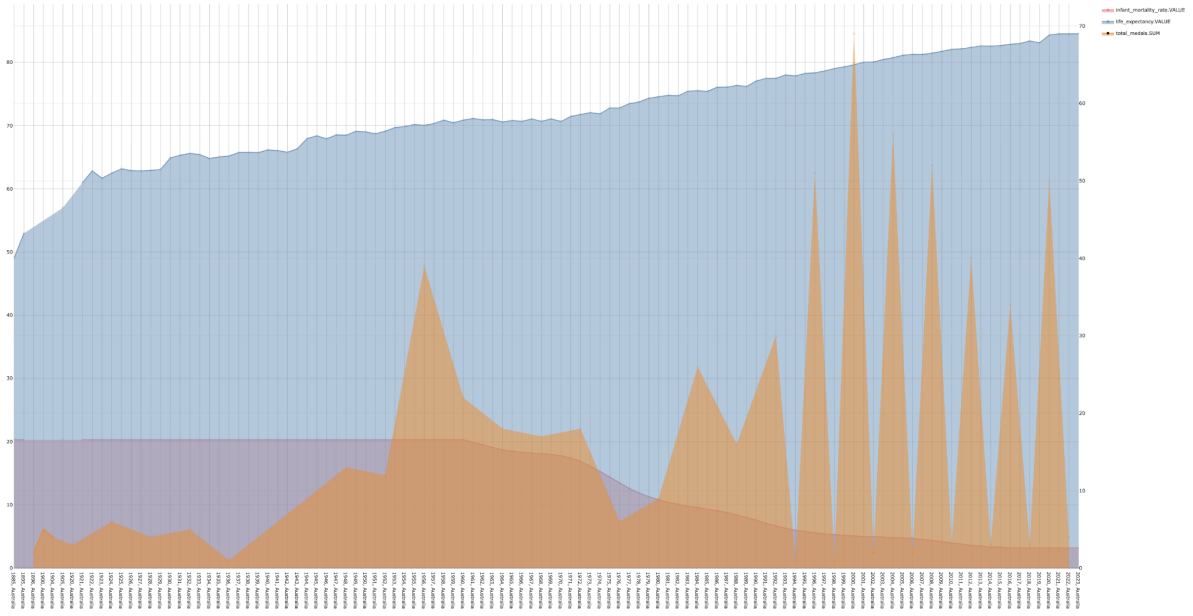


Figure 36. Life expectancy, infant mortality rate and Olympic medals

As life expectancy has risen and infant mortality has decreased, there is a corresponding increase in the number of Olympic medals. This trend suggests a potential correlation between improvements in public health indicators and enhanced athletic performance over time. It's plausible that advancements in healthcare lead to healthier and more physically capable populations, which may contribute to better athletic outcomes at the Olympic Games.

There are sudden downward spike in the diagram which are actually the number of medals in the Winter games. Australia's relatively lower count of Winter Olympics medals compared to Summer Olympics medals can be attributed to several factors. The country's climate and geography favor summer sports, and historical focus and infrastructure have prioritized summer sports development. Additionally, the late development of winter sports programs, smaller population, and less cultural interest in winter sports contribute to this disparity.

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

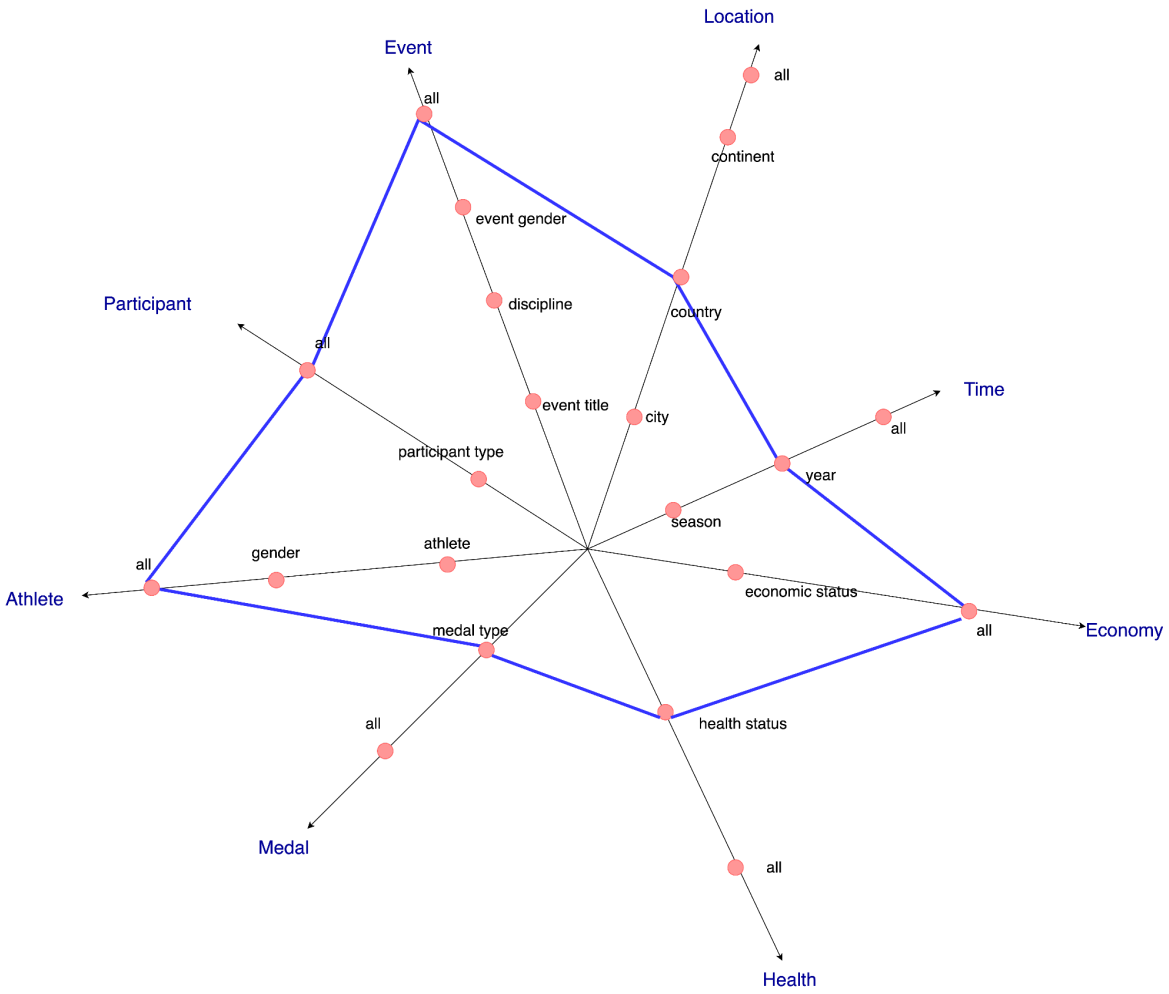


Figure 39. Starnet Query Model - Comparative Health Analysis:

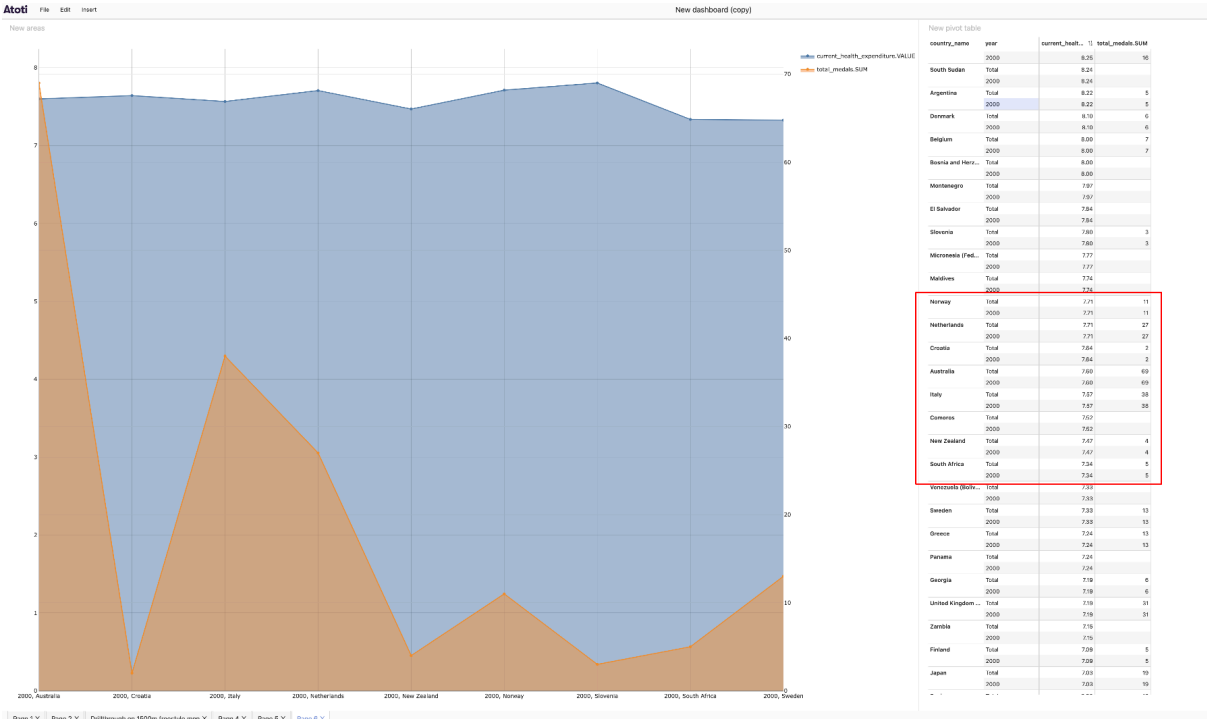


Figure 38. Comparative Health Analysis:

Despite similar health expenditures in 2000 among Norway, Netherlands, New Zealand, South Africa, and Italy, there were significant differences in the number of medals won by these countries. This indicates that health expenditures alone do not solely determine the chances of winning medals.

Argumentative Essay: The Relevance of Data Cubes in Modern Data Warehousing

Some articles argue that "The data cube is an outdated technology."

For example, in a 2023 article, Albert Wong argued that "database cubes were popular in the early days of data warehousing, but they have largely been replaced by other technologies."
Do you agree or disagree with this point? If you disagree with this point, discuss your reason. If you agree with this point, discuss your reasons and explain at least one technology that can replace the data cube.

Data cubes have long been a cornerstone in the field of data warehousing, providing a multi-dimensional structure for data analysis and facilitating efficient querying and reporting. However, the dynamic evolution of data technologies suggests a reassessment of their relevance in the modern data landscape.

The assertion that data cubes are becoming outdated is grounded in significant technological advancements and shifts in data handling strategies. Traditional data cubes excel in scenarios with static schemas and predictable querying patterns. However, the modern data ecosystem, characterized by vast volumes of unstructured data, real-time processing needs, and cloud-based storage solutions, demands more flexible and scalable systems.

Data cubes, while efficient for predefined queries, struggle with the scalability demands posed by Big Data. As businesses generate more complex and larger datasets, the rigid structure of data cubes becomes a bottleneck, unable to adapt quickly to changing data types or new business questions. Technologies like Apache Spark have gained prominence by offering robust in-memory processing capabilities, which allow for faster data manipulation and analysis without the pre-aggregation required by traditional OLAP cubes. This not only speeds up the analytical processes but also supports a wider variety of data formats and complex computational algorithms. This provides a more flexible architecture suited to unstructured data, which is on the rise. These technologies support diverse data types and the integration of machine learning algorithms, which are increasingly crucial for gaining actionable insights.

Data lakes represent a crucial technological advancement poised to replace traditional data cubes. Unlike data cubes that require data to be fitted into predefined dimensions, data lakes store raw data in its native format, which includes structured, semi-structured, and unstructured data. This approach not only eliminates the need for extensive upfront data modelling but also provides the agility to adapt to diverse analytical demands. Additionally, data lakes integrate seamlessly with modern, distributed computing platforms like Hadoop and cloud services, offering scalability and flexibility that traditional data warehousing struggles to match.

At the same time, industry trends indicate a significant shift towards real-time analytics, where data latency can hinder business decision-making processes. Modern businesses

require agile data environments that can respond instantly to market changes and customer needs. This need for speed and flexibility is something that traditional data cubes, with their batch-processing orientation, cannot fulfil effectively. Thus, integrating advanced technologies like data lakes with cutting-edge analytics tools and artificial intelligence further enhances the capability to perform complex data analysis in real-time, driving better business outcomes and maintaining a competitive edge in the market.

While data cubes have served as a fundamental component of data warehousing frameworks, their utility in the face of rapidly advancing technological landscapes is diminishing. The shift towards more adaptable, scalable, and powerful technologies such as data lakes signifies a critical evolution in data management strategies. Embracing these new technologies will enable organisations to harness the full potential of their data assets and stay competitive in a data-driven world.

References:

- ChatGPT has been used to generate some codes in the process and get insights on how to tackle some debugging issues.
- https://docs.atoti.io/latest/api/atoti.html
- https://www.nbcnewyork.com/news/sports/beijing-winter-olympics/history-of-united-states-cities-hosting-the-olympics/3209503/
- **https://atwong.medium.com/database-cubes-are-dead-what-is-their-replacement-999a0014f32c**
- **https://www.sigmacomputing.com/blog/the-decline-of-the-business-intelligence-cube-and-whats-replaced-it**
- **https://www.holistics.io/blog/the-rise-and-fall-of-the-olap-cube/**

Appendix

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]:
```
import pandas as pd
```

```
# Load the CSV file
file_path = './raw_data/economic_data_external.csv'
economic_data = pd.read_csv(file_path)

# Display the first few rows and the data types of each column
economic_data.head()
```

Out[2]:

	Time	Time Code	Country Name	Country Code	GDP per capita (current US$) [NY.GDP.PCAP.CD]	GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]	Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]	Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
0	1960	YR1960	Argentina	ARG	..	..	..	..
1	1960	YR1960	Australia	AUS	1810.59744278609	..	..	..
2	1960	YR1960	Brazil	BRA	..	..	..	..
3	1960	YR1960	China	CHN	89.5202142368479	..	..	..
4	1960	YR1960	France	FRA	1333.88157286682	..	..	..

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]:
```
# data = data.drop(columns=['Time Code', 'Country Name'])
economic_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17024 entries, 0 to 17023
Data columns (total 13 columns):
 #   Column
          Non-Null Count  Dtype
---  ------
          --------------  -----
 0   Time
```

```
          17024 non-null  int64
 1   Time Code
          17024 non-null  object
 2   Country Name
          17024 non-null  object
 3   Country Code
          17024 non-null  object
 4   GDP per capita (current US$) [NY.GDP.PCAP.CD]
          17024 non-null  object
 5   GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]
          17024 non-null  object
 6   Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]
          17024 non-null  object
 7   Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
          17024 non-null  object
 8   Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]
          17024 non-null  object
 9   Domestic general government health expenditure per capita (current US$) [SH.XPD.GHE
D.PC.CD]  17024 non-null  object
 10  Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]
          17024 non-null  object
 11  External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]
          17024 non-null  object
 12  Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]
          17024 non-null  object
dtypes: int64(1), object(12)
memory usage: 1.7+ MB
```

In [4]:

```python
# Convert columns that are intended to be numeric but are currently object type, except
numeric_columns = economic_data.columns[4:]  # Assuming first four columns are categoric

# Replace '..' with NaN and convert columns to float
for col in numeric_columns:
    economic_data[col] = pd.to_numeric(economic_data[col], errors='coerce')

# Check the conversion
economic_data.dtypes
```

Out[4]:

```
Time
         int64
Time Code
        object
Country Name
        object
Country Code
        object
GDP per capita (current US$) [NY.GDP.PCAP.CD]
       float64
GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]
       float64
Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]
       float64
Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
       float64
Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]
       float64
Domestic general government health expenditure per capita (current US$) [SH.XPD.GHED.PC.
CD]    float64
Domestic private health expenditure per capita (current US$) [SH.XPD.PVTD.PC.CD]
       float64
External health expenditure per capita (current US$) [SH.XPD.EHEX.PC.CD]
       float64
Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]
       float64
dtype: object
```

In [5]:
```
economic_data.head()
```

Out[5]:

	Time	Time Code	Country Name	Country Code	GDP per capita (current US$) [NY.GDP.PCAP.CD]	GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]	Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]	Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
0	1960	YR1960	Argentina	ARG	NaN	NaN	NaN	NaN
1	1960	YR1960	Australia	AUS	1810.597443	NaN	NaN	NaN
2	1960	YR1960	Brazil	BRA	NaN	NaN	NaN	NaN
3	1960	YR1960	China	CHN	89.520214	NaN	NaN	NaN
4	1960	YR1960	France	FRA	1333.881573	NaN	NaN	NaN

In [6]:
```
# Group by 'Country Name' and apply interpolation within each group
economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group: g

# Display the first few rows after grouped interpolation to verify changes
economic_data_interpolated
```

```
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
```

```
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
 group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
 group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
 group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
 group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
 group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
 group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
 group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93740/4103772907.py:2: DeprecationWarning: DataFrameGroupBy.apply operate
d on the grouping columns. This behavior is deprecated, and in a future version of panda
s the grouping columns will be excluded from the operation. Either pass `include_groups=
False` to exclude the groupings or explicitly select the grouping columns after groupby
 to silence this warning.
  economic_data_interpolated = economic_data.groupby('Country Code').apply(lambda group:
 group.interpolate(method='linear', limit_direction='both'))
```

Out[6]:

Country Code		Time	Time Code	Country Name	Country Code	GDP per capita (current US$) [NY.GDP.PCAP.CD]	GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]	Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]	[I
ABW	**28**	1960	YR1960	Aruba	ABW	6283.001443	16.263941	NaN	
	294	1961	YR1961	Aruba	ABW	6283.001443	16.263941	NaN	
	560	1962	YR1962	Aruba	ABW	6283.001443	16.263941	NaN	
	826	1963	YR1963	Aruba	ABW	6283.001443	16.263941	NaN	
	1092	1964	YR1964	Aruba	ABW	6283.001443	16.263941	NaN	
...	**...**	...	...	...	...	...	...	...	
ZWE	**15910**	2019	YR2019	Zimbabwe	ZWE	1421.868596	-8.177320	39.8	
	16176	2020	YR2020	Zimbabwe	ZWE	1372.696674	-9.670405	39.8	
	16442	2021	YR2021	Zimbabwe	ZWE	1773.920411	6.271613	39.8	

16708	2022	YR2022	Zimbabwe	ZWE	1676.821489	4.387997	39.8
16974	2023	YR2023	Zimbabwe	ZWE	1676.821489	4.387997	39.8

17024 rows × 13 columns

Upon exploring the interpolated data, Because some countries does not have any data regarding some columns, we will fill them with medians

In [7]:
```
# Calculate the median for each column and fill missing values with the median
columns_with_na = economic_data_interpolated.columns[economic_data_interpolated.isna().a
medians = economic_data_interpolated[columns_with_na].median()

# Fill missing values with medians
economic_data_interpolated_filled = economic_data_interpolated.fillna(medians)

# Check the summary of missing values after filling to confirm
economic_data_interpolated_filled.isna().sum()
```

Out[7]:
```
Time
       0
Time Code
       0
Country Name
       0
Country Code
       0
GDP per capita (current US$) [NY.GDP.PCAP.CD]
       0
GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]
       0
Poverty headcount ratio at $2.15 a day (2017 PPP) (% of population) [SI.POV.DDAY]
       0
Secure Internet servers (per 1 million people) [IT.NET.SECR.P6]
       0
Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]
       0
Domestic general government health expenditure per capita (current US$) [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
Mortality rate, infant (per 1,000 live births) [SP.DYN.IMRT.IN]
       0
dtype: int64
```

Economic Data Cleaning Update

Updated Missing Values

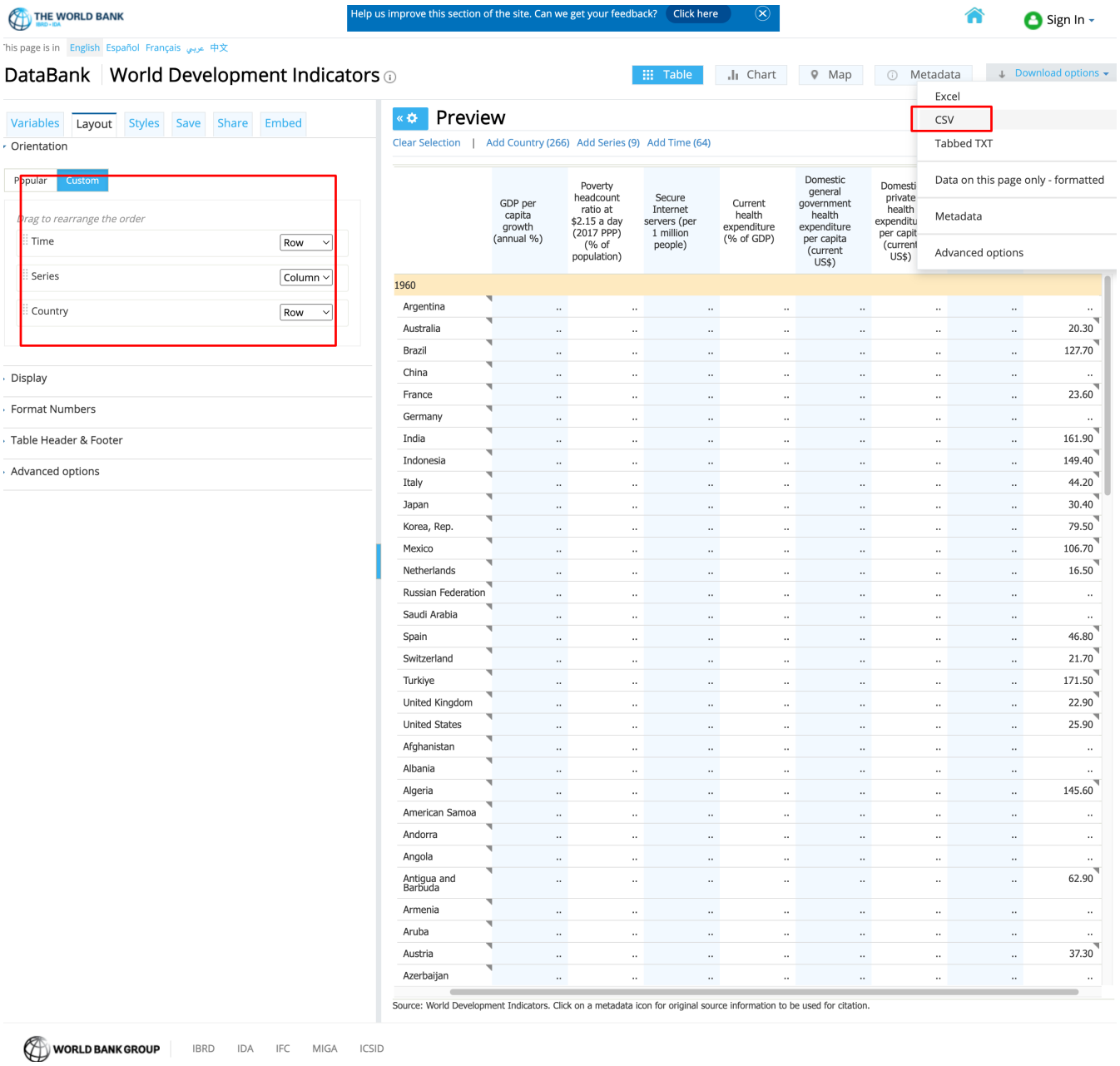
After cleaning, here's the updated count of missing values per column:

- **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.

The data used to address client queries was sourced from The World Bank, specifically from the World Development Indicators database. The original dataset provided only included information for the year 2020. To enhance the analysis and support more comprehensive business inquiries, additional data spanning various years and countries was downloaded. The layout of this expanded dataset is illustrated in the screenshot provided. This enriched dataset will be utilized for various business-related analyses.



Loading economic data to OLTP database

The next step I took was to load the cleaned data into an OLTP (Online Transaction Processing) system to facilitate querying during the ETL (Extract, Transform, Load) process. I created a database named `olympic_oltp` to house all the cleaned data. This setup will support efficient data retrieval and management throughout the ETL operations.

In [8]:
```
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 hav
    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 cr

    # 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 [9]:
```
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 [10]:
```
# 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 [11]:
```
# Rename the columns to match the PostgreSQL table schema exactly

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

In [12]:
```
economic_data_interpolated_filled
```

Out[12]:

Country Code		time_year	time_code	country_name	country_code	gdp_per_capita_usd	gdp_per_capita_growt
ABW	**28**	1960	YR1960	Aruba	ABW	6283.001443	16.26394
	294	1961	YR1961	Aruba	ABW	6283.001443	16.26394
	560	1962	YR1962	Aruba	ABW	6283.001443	16.26394
	826	1963	YR1963	Aruba	ABW	6283.001443	16.26394
	1092	1964	YR1964	Aruba	ABW	6283.001443	16.26394
...	**...**	...	...	...	...	...	.
ZWE	**15910**	2019	YR2019	Zimbabwe	ZWE	1421.868596	-8.17732
	16176	2020	YR2020	Zimbabwe	ZWE	1372.696674	-9.67040
	16442	2021	YR2021	Zimbabwe	ZWE	1773.920411	6.27161
	16708	2022	YR2022	Zimbabwe	ZWE	1676.821489	4.38799
	16974	2023	YR2023	Zimbabwe	ZWE	1676.821489	4.38799

17024 rows × 13 columns

In [13]:
```
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 pe
    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 ca
    private_health_expenditure_per_capita_usd FLOAT, -- Private health expenditure per c
    external_health_expenditure_per_capita_usd FLOAT -- External health expenditure per
);
"""

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 [14]:
```
economic_data_interpolated_filled.to_sql("economic_data", con=engine, if_exists="append"
```

Out[14]:
```
24
```

In [15]:
```
# Save the cleaned data
economic_data_interpolated_filled.to_csv('cleaned_data/economic_data_external.csv', inde
```

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 [16]:
```
global_population_data = pd.read_csv('./raw_data/Global Population.csv', encoding='ISO-8
```

In [17]:
```
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
 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 [18]:
```
global_population_data_missing_values
```

Out[18]:
```
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 [19]:
```
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 N
    global_population_data[col] = pd.to_numeric(global_population_data[col], errors='coe
```

- **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 [20]:
```
global_population_data.head()
```

Out[20]:

	Population (Millions of people)	1980	1981	1982	1983	1984	1985	1986	1987	1988	...	2019	2020	202
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	Na
1	Afghanistan	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	32.200	32.941	33.69
2	Albania	2.672	2.726	2.784	2.844	2.904	2.965	3.023	3.084	3.142	...	2.881	2.878	2.87
3	Algeria	18.666	19.246	19.864	20.516	21.175	22.200	22.800	23.400	24.100	...	43.424	43.851	44.57
4	Andorra	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	0.078	0.078	0.08

5 rows × 50 columns

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

In [21]:
```
# Interpolate missing data for each country
global_population_data.iloc[:, 1:] = global_population_data.iloc[:, 1:].apply(lambda x:
```

In [22]:
```
global_population_data = global_population_data.rename(columns={'Population (Millions of
```

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

In [23]:
```
# 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 [24]:
```
global_population_data
```

Out[24]:

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

228 rows × 50 columns

In [25]:
```
global_population_data.columns
```

Out[25]:
```
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 [26]:
```
# 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,
    "1987" FLOAT, "1988" FLOAT, "1989" FLOAT, "1990" FLOAT, "1991" FLOAT, "1992" FLOAT,
    "1994" FLOAT, "1995" FLOAT, "1996" FLOAT, "1997" FLOAT, "1998" FLOAT, "1999" FLOAT,
    "2001" FLOAT, "2002" FLOAT, "2003" FLOAT, "2004" FLOAT, "2005" FLOAT, "2006" FLOAT,
    "2008" FLOAT, "2009" FLOAT, "2010" FLOAT, "2011" FLOAT, "2012" FLOAT, "2013" FLOAT,
    "2015" FLOAT, "2016" FLOAT, "2017" FLOAT, "2018" FLOAT, "2019" FLOAT, "2020" FLOAT,
    "2022" FLOAT, "2023" FLOAT, "2024" FLOAT, "2025" FLOAT, "2026" FLOAT, "2027" FLOAT,
```

```
);
"""
```

In [27]:
```
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 [28]:
```
global_population_data.to_sql("population_data", con=engine, if_exists="append", index=F
```

Out[28]:
```
228
```

In [29]:
```
# Save the cleaned data
global_population_data.to_csv('cleaned_data/Global Population.csv', index=False)
```

3. Life Expectancy

In [30]:
```
life_expectancy_path = './raw_data/life-expectancy.csv'

life_expectancy_data = pd.read_csv(life_expectancy_path)
```

In [31]:
```
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 [32]:
```
life_expectancy_data.head()
```

Out[32]:

	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 [33]:
```
life_expectancy_data.columns
```

Out[33]:
```
Index(['Entity', 'Code', 'Year',
       'Period life expectancy at birth - Sex: all - Age: 0'],
      dtype='object')
```

In [ ]:

In [34]:
```
# 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 b
```

In [35]:
```
life_expectancy_data.columns
```

Out[35]:
```
Index(['entity', 'country_code', 'year', 'life_expectancy'], dtype='object')
```

In [36]:
```
# 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 [37]:
```
life_expectancy_data.to_sql("life_expectancy_data", con=engine, if_exists="append", inde
```

Out[37]:
```
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 [38]:
```
# 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
```

In [39]:
```
# 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 [40]:
```
# 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[40]:

	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
3	Afghanistan	AFG	1993	891.51587	136.48602	146.78812	23.863169	436.69104
4	Afghanistan	AFG	1994	891.39160	136.18323	146.58481	23.189074	436.76800

In [41]:
```
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-standard
ized  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-standardize
d     6840 non-null   float64
dtypes: float64(5), int64(1), object(2)
memory usage: 427.6+ KB
```

```
mental_illness_data.columns
```

In [42]:

Out[42]:
```
Index(['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'],
      dtype='object')
```

In [43]:
```
mental_illness_data.columns = [
    'entity',
    'country_code',
    'year',
    'daly_depression',
    'daly_schizophrenia',
    'daly_bipolar_disorder',
    'daly_eating_disorder',
    'daly_anxiety'
]
```

In [44]:
```
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   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 [45]:
```
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 [46]:
```
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 [47]:
```
mental_illness_data.to_sql("mental_health_data", con=engine, if_exists="append", index=F
```

Out[47]:
```
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 [48]:
```
# 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' c

# 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[48]:
```
('./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.

```
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 [49]:
```
olympic_host_data_path = './raw_data/olympic_hosts.csv'

olympic_host_data = pd.read_csv(olympic_host_data_path)
```

In [50]:
```
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 [51]:
```
olympic_host_data.columns
```

Out[51]:
```
Index(['game_slug', 'game_end_date', 'game_start_date', 'game_location',
       'game_name', 'game_season', 'game_year'],
      dtype='object')
```

In [52]:
```
from datetime import datetime

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

In [53]:
```
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 [54]:
```
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 [55]:
```
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 [56]:
```
olympic_host_data.to_sql("olympic_hosts", con=engine, if_exists="append", index=False)
```

Out[56]:
```
53
```

6. Olympic Medals

In [57]:
```
# 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 [58]:
```
olympic_medals_data.head()
```

Out[58]:

	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://oly
1	Curling	beijing-2022	Mixed Doubles	Mixed	GOLD	GameTeam	Italy	https://
2	Curling	beijing-2022	Mixed Doubles	Mixed	SILVER	GameTeam	Norway	https://
3	Curling	beijing-2022	Mixed Doubles	Mixed	SILVER	GameTeam	Norway	https://oly
4	Curling	beijing-2022	Mixed Doubles	Mixed	BRONZE	GameTeam	Sweden	https://o

In [59]:
```
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 [60]:
```
olympic_medals_data_missing_values = olympic_medals_data.isnull().sum()
```

In [61]:
```
olympic_medals_data_missing_values
```

Out[61]:
```
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 [62]:
```
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 [63]:
```
connection = create_connection(db_name, db_user, db_password, db_host, db_port)
cursor = connection.cursor()
```

```
Connection to PostgreSQL DB successful
```

In [64]:
```
cursor.execute(create_olympic_medals_table_query)
connection.commit()  # Commit the changes
cursor.close()
connection.close()
```

In [65]:
```
olympic_medals_data.to_sql("olympic_medals", con=engine, if_exists="append", index=False
```

Out[65]:
```
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 [66]:
```
# 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[66]:

	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 [67]:
```
countries_df = countries_df.drop(['color', 'id', 'M49 code', 'region'], axis=1)
```

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   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 [69]:
```
countries_df.columns = [ "country_name", "country_code", "continent"]
```

In [70]:
```
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 [71]:
```
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 [72]:
```
connection = create_connection(db_name, db_user, db_password, db_host, db_port)
cursor = connection.cursor()
```

```
Connection to PostgreSQL DB successful
```

In [73]:
```
cursor.execute(create_countries_table_query)
connection.commit()  # Commit the changes
cursor.close()
connection.close()
```

In [74]:
```
countries_df.to_sql("countries", con=engine, if_exists="append", index=False)
```

Out[74]:
```
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 hav
    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 cr

    # 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
```

```
);
""")
```

In [15]:
```
# Create CombinedFactTable

olap_cursor.execute("""
CREATE TABLE CombinedFactTable (
    id SERIAL PRIMARY KEY,
    game_slug VARCHAR(100),
    participant_id INTEGER,
    athlete_id INTEGER,
    event_id INTEGER,
    country_code CHAR(3),
    year INTEGER,
    total_bronze_medals INTEGER,
    total_silver_medals INTEGER,
    total_gold_medals INTEGER,
    total_medals INTEGER,
    poverty_count FLOAT,
    gdp_per_capita FLOAT,
    annual_gdp_growth FLOAT,
    servers_count INTEGER,
    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,
    FOREIGN KEY (game_slug) REFERENCES DimGame(game_slug),
    FOREIGN KEY (participant_id) REFERENCES DimParticipant(participant_id),
    FOREIGN KEY (athlete_id) REFERENCES DimAthlete(athlete_id),
    FOREIGN KEY (event_id) REFERENCES DimEvent(event_id),
    FOREIGN KEY (country_code) REFERENCES DimLocation(country_code),
    FOREIGN KEY (year) REFERENCES DimYear(year)
);
""")
```

In [16]:
```
olap_connection.commit()
```

Loading data to Dimension tables

1. DimYear

In [17]:
```
# Insert values from 1543 (life expectancy data) - data in the available record to 2028

olap_cursor.execute("""
INSERT INTO DimYear (year)
SELECT generate_series AS year
FROM generate_series(1543, 2028);
""")

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 [18]:
```
oltp_connection = create_connection('olympic_oltp', db_user, db_password, db_host, db_po

oltp_cursor = oltp_connection.cursor()
```

```
Connection to PostgreSQL DB successful
```

In [19]:
```
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[19]:

	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	Athletics	Men
4	10000m men	Speed skating	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 [20]:
```
from sqlalchemy import create_engine

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

# 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[20]:
```
588
```

3. DimParticipant

In [21]:
```
# Execute the SQL query using the oltp_cursor

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

# Fetch all the unique rows returned by the query
dim_participant_data = oltp_cursor.fetchall()
```

In [22]:
```
# Extract column names from the cursor description attribute
# Each item in 'description' contains metadata about each column used in the SQL result,
# 'desc[0]' accesses the name of the column

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

Out[22]:
```
['participant_title', 'participant_type']
```

In [23]:
```
dim_participant_df = pd.DataFrame(dim_participant_data, columns = dim_participant_data_c
dim_participant_df.head()
```

Out[23]:

	participant_title	participant_type
0	Romania team	GameTeam
1	Tan-Fe-Pah	GameTeam
2	Pupilles de Neptune de Lille #1	GameTeam
3	Slovenia team	GameTeam
4	Gallia II	GameTeam

In [24]:
```
dim_participant_df.to_sql("dimparticipant", con=olap_engine, if_exists="append", index=F
```

Out[24]:
```
494
```

4. DimAthlete

I have repeated similar process for all the dimension tables below

In [25]:
```
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 [26]:
```
dim_athlete_data_columns = [desc[0] for desc in oltp_cursor.description]
dim_athlete_data_columns
```

Out[26]:
```
['athlete_name', 'athlete_url']
```

In [27]:
```
dim_athlete_df = pd.DataFrame(dim_athlete_data, columns = dim_athlete_data_columns)
dim_athlete_df.head()
```

Out[27]:

	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 [28]:
```
dim_athlete_df.to_sql("dimathlete", con=olap_engine, if_exists="append", index=False)
```

Out[28]:
```
116
```

5. DimLocation

This stage represents the most comprehensive aspect of the ETL (Extract, Transform, Load) process, due to numerous discrepancies in naming conventions. To address this, I have adopted standardized names and new country codes, as detailed in the OLTP notebook. The approach involves replacing any non-standard names with the standardized equivalents from a reference table. This strategy aims to streamline the process, ensuring consistency across the data and minimizing issues related to data duplication and deletion. All analytical steps and comparisons undertaken to achieve this standardization are thoroughly documented in the accompanying documentation.

Link to the Mismatched country names

In [29]:
```
oltp_cursor.execute("""
SELECT * FROM countries;
""")

dim_location_data = oltp_cursor.fetchall()
```

In [30]:
```
dim_location_data_columns = [desc[0] for desc in oltp_cursor.description]
dim_location_data_columns
```

Out[30]:
```
['country_code', 'country_name', 'continent']
```

In [31]:
```
dim_location_df = pd.DataFrame(dim_location_data, columns = dim_location_data_columns)
dim_location_df.head()
```

Out[31]:

	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 [32]:
```
dim_location_df.to_sql("dimlocation", con=olap_engine, if_exists="append", index=False)
```

Out[32]:
```
248
```

When exploring discrepancies in country names, I executed the following SQL query to identify how names differ between the `olympic_hosts` file and our standard `countries` database:

```
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;
```

This query helped identify several cases where country names in the `olympic_hosts` file did not match the standardized names in the `countries` table. The discrepancies found and the corresponding standardized names are as follows:

- "Australia, Sweden" is corrected to "Australia"
- "Federal Republic of Germany" is updated to "Germany"
- "Great Britain" is standardized as "United Kingdom of Great Britain and Northern Ireland"
- "United States" is referred to as "United States of America"
- "USSR" is updated to "Russian Federation"
- "Yugoslavia" is changed to "Serbia"

These findings necessitate updating the `olympic_hosts` file to align with the standard names used in the `countries` database to ensure data consistency.

In [33]:
```
oltp_cursor.execute("""
CREATE TABLE olympic_hosts_backup AS
SELECT *
FROM olympic_hosts;
""")

oltp_connection.commit()
```

In [34]:
```
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 North
    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 [35]:
```
oltp_cursor.execute("""
CREATE TABLE olympic_medals_backup AS
SELECT *
FROM olympic_medals;
""")

oltp_connection.commit()
```

The country names are included in nearly every CSV file I used. To ensure consistency across the database, I created a standardized table for country names. A sample of this table is mentioned below, and more detailed information can be found in the link provided earlier.

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 - 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	
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 [36]:
```
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 [37]:
```
oltp_connection.commit()
```

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

In [38]:
```
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 [39]:
```
oltp_cursor.execute("""
DELETE FROM olympic_medals
WHERE country_3_letter_code IN ('ANZ', 'IOA', 'MIX', 'WIF', 'XKX');
""")


oltp_connection.commit()
```

Executing the SQL query below helped identify mismatches between country codes in the `life_expectancy_data` and the `countries` tables:

```
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;
```

This query revealed discrepancies and required actions:

- "OWID_WRL" which represents the world, should be removed from the dataset.
- "OWID_KOS", representing Kosovo, is not recognized in the `countries` table.
- "OWID_USS" should be updated to "RUS" for Russia.

Before performing any operation, I started creating backup first.

a. life expectancy

I updated the mismatched country codes in the life expectancy dataset.

In [40]:
```
oltp_cursor.execute("""
CREATE TABLE life_expectancy_data_backup AS
SELECT *
FROM life_expectancy_data;
""")

oltp_connection.commit()
```

In [41]:
```
oltp_cursor.execute("""
-- Delete rows where the country code is 'OWID_WRL' (World) or 'OWID_KOS' (Kosovo)
DELETE FROM life_expectancy_data
WHERE country_code = 'OWID_WRL' OR country_code = 'OWID_KOS';
""")

# Execute the SQL command to delete rows where the country code is null
oltp_cursor.execute("""
DELETE FROM life_expectancy_data
WHERE country_code IS NULL;
""")

# Execute the SQL command to update the country code
oltp_cursor.execute("""
UPDATE life_expectancy_data
SET country_code = 'RUS'
WHERE country_code = 'OWID_USS';
""")
```

```
oltp_connection.commit()
```

b. mental health data

I removed the `OWID_WRL` entry since it represents global data, which isn't necessary for our analysis that focuses on individual countries. If needed, we can aggregate the data from all countries to derive global statistics.

In [42]:
```
# Create a backup of the mental_health_data table
oltp_cursor.execute("""
CREATE TABLE mental_health_data_backup AS
SELECT *
FROM mental_health_data;
""")

# Delete records from the mental_health_data table where the country code is either null
oltp_cursor.execute("""
DELETE FROM mental_health_data
WHERE country_code IS NULL OR country_code = 'OWID_WRL';
""")

oltp_connection.commit()
```

c. population data

I removed data pertaining to regions and retained only the information specific to individual countries.

In [43]:
```
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'
);
""")
```

I utilized the TRIM function to remove trailing whitespaces from names, ensuring accurate matching during name comparisons.

In [44]:
```
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
    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 Administrat
    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 Regi
    WHEN TRIM("Population") = 'Micronesia, Fed. States of' THEN 'Micronesia (Federated S
    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
    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 [45]:
```
oltp_cursor.execute("""
-- Creating a backup of the economic_data table
CREATE TABLE economic_data_backup AS
SELECT *
FROM economic_data;
""")
```

For consistency, we will map the old country names to new country codes using the table above

In [46]:
```
oltp_cursor.execute("""
UPDATE economic_data
SET country_code = CASE
    WHEN country_code = 'AHO' THEN 'NLD'
```

```
    WHEN country_code = 'ALG' THEN 'DZA'
    WHEN country_code = 'BAH' THEN 'BHS'
    WHEN country_code = 'BAR' THEN 'BRB'
    WHEN country_code = 'BER' THEN 'BMU'
    WHEN country_code = 'BOH' THEN 'CZE'
    WHEN country_code = 'BOT' THEN 'BWA'
    WHEN country_code = 'BUL' THEN 'BGR'
    WHEN country_code = 'BUR' THEN 'MMR'
    WHEN country_code = 'CHI' THEN 'CHL'
    WHEN country_code = 'CRC' THEN 'CRI'
    WHEN country_code = 'CRO' THEN 'HRV'
    WHEN country_code = 'DEN' THEN 'DNK'
    WHEN country_code = 'FIJ' THEN 'FJI'
    WHEN country_code = 'FRG' THEN 'DEU'
    WHEN country_code = 'GDR' THEN 'DEU'
    WHEN country_code = 'GER' THEN 'DEU'
    WHEN country_code = 'GRE' THEN 'GRC'
    WHEN country_code = 'GRN' THEN 'GRD'
    WHEN country_code = 'GUA' THEN 'GTM'
    WHEN country_code = 'HAI' THEN 'HTI'
    WHEN country_code = 'INA' THEN 'IDN'
    WHEN country_code = 'IRI' THEN 'IRN'
    WHEN country_code = 'ISV' THEN 'VIR'
    WHEN country_code = 'KOS' THEN 'XKX'
    WHEN country_code = 'KSA' THEN 'SAU'
    WHEN country_code = 'KUW' THEN 'KWT'
    WHEN country_code = 'LAT' THEN 'LVA'
    WHEN country_code = 'MAS' THEN 'MYS'
    WHEN country_code = 'MGL' THEN 'MNG'
    WHEN country_code = 'MRI' THEN 'MUS'
    WHEN country_code = 'NED' THEN 'NLD'
    WHEN country_code = 'NGR' THEN 'NGA'
    WHEN country_code = 'NIG' THEN 'NER'
    WHEN country_code = 'PAR' THEN 'PRY'
    WHEN country_code = 'PHI' THEN 'PHL'
    WHEN country_code = 'POR' THEN 'PRT'
    WHEN country_code = 'PUR' THEN 'PRI'
    WHEN country_code = 'ROC' THEN 'TWN'
    WHEN country_code = 'RSA' THEN 'ZAF'
    WHEN country_code = 'SAM' THEN 'WSM'
    WHEN country_code = 'SCG' THEN 'SRB'
    WHEN country_code = 'SLO' THEN 'SVN'
    WHEN country_code = 'SRI' THEN 'LKA'
    WHEN country_code = 'SUD' THEN 'SDN'
    WHEN country_code = 'SUI' THEN 'CHE'
    WHEN country_code = 'TAN' THEN 'TZA'
    WHEN country_code = 'TCH' THEN 'CZE'
    WHEN country_code = 'TGA' THEN 'TON'
    WHEN country_code = 'TOG' THEN 'TGO'
    WHEN country_code = 'TPE' THEN 'TWN'
    WHEN country_code = 'UAE' THEN 'ARE'
    WHEN country_code = 'UAR' THEN 'EGY'
    WHEN country_code = 'URS' THEN 'RUS'
    WHEN country_code = 'URU' THEN 'URY'
    WHEN country_code = 'VIE' THEN 'VNM'
    WHEN country_code = 'YUG' THEN 'SRB'
    WHEN country_code = 'ZAM' THEN 'ZMB'
    WHEN country_code = 'ZIM' THEN 'ZWE'
    ELSE country_code
END;
""")
```

```
SELECT DISTINCT e.country_code
FROM economic_data e
```

```
LEFT JOIN countries c ON c.country_code = e.country_code
WHERE c.country_code IS NULL;
```

The following country code is not included in our standard list of countries, so we will remove all rows associated with this country code. | country_code | |----------| | PST | | TEC | | TLA | | UMC | | OED | | ARB | | HIC | | LMC | | AFW | | MNA | | MEA | | IBT | | CEB | | MIC | | PRE | | LAC | | EUU | | SSA | | INX | | OSS | | LIC | | SSF | | TMN | | HPC | | TEA | | ECA | | EMU | | IBD | | LCN | | PSS | | LMY | | LDC | | NAC | | TSA | | WLD | | LTE | | SAS | | IDA | | ECS | | CSS | | AFE | | IDB | | SST | | FCS | | EAR | | EAS | | TSS | | EAP | | IDX |

In [47]:
```
oltp_cursor.execute("""
DELETE FROM economic_data
WHERE country_code IN (
    'PST', 'TEC', 'TLA', 'UMC', 'OED', 'ARB', 'HIC', 'LMC', 'AFW', 'MNA',
    'MEA', 'IBT', 'CEB', 'MIC', 'PRE', 'LAC', 'EUU', 'SSA', 'INX', 'OSS',
    'LIC', 'SSF', 'TMN', 'HPC', 'TEA', 'ECA', 'EMU', 'IBD', 'LCN', 'PSS',
    'LMY', 'LDC', 'NAC', 'TSA', 'WLD', 'LTE', 'SAS', 'IDA', 'ECS', 'CSS',
    'AFE', 'IDB', 'SST', 'FCS', 'EAR', 'EAS', 'TSS', 'EAP', 'IDX', 'XKX'
);
""")
```

In [48]:
```
oltp_connection.commit()
```

6. DimGame

I replaced country names with country codes in the Hosts information. This adjustment will facilitate the creation of fact tables and the construction of cubes, enhancing the consistency and efficiency of data management.

In [49]:
```
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 [50]:
```
dim_game_data_columns = [desc[0] for desc in oltp_cursor.description]
dim_game_data_columns
```

Out[50]:
```
['game_slug', 'game_name', 'game_season', 'game_year', 'country_code']
```

In [51]:
```
dim_game_df =  pd.DataFrame(dim_game_data, columns = dim_game_data_columns)

dim_game_df.head()
```

Out[51]:

	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
3	rio-2016	Rio 2016	Summer	2016	BRA
4	sochi-2014	Sochi 2014	Winter	2014	RUS

In [52]:
```
dim_game_df.to_sql("dimgame", con=olap_engine, if_exists="append", index=False)
```

Out[52]:
```
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 [53]:
```
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 [54]:
```
olap_olympic_data = oltp_cursor.fetchall()
```

In [55]:
```
olap_olympic_data_columns = [desc[0] for desc in oltp_cursor.description]
olap_olympic_data_columns
```

Out[55]:
```
['discipline_title',
 'slug_game',
 'event_title',
 'event_gender',
 'medal_type',
 'participant_type',
 'participant_title',
 'athlete_full_name',
 'country_code']
```

In [56]:
```
olap_olympic_df =  pd.DataFrame(olap_olympic_data, columns = olap_olympic_data_columns)
```

In [57]:
```
olap_olympic_df.head()
```

Out[57]:

	discipline_title	slug_game	event_title	event_gender	medal_type	participant_type	participant_title	athlete_f
0	Football	tokyo-2020	Women	Women	GOLD	GameTeam	Canada	
1	Archery	tokyo-2020	Women's Team	Women	SILVER	GameTeam	ROC	
2	Softball	beijing-2008	softball women	Women	GOLD	GameTeam	Japan team	
3	Football	seoul-1988	football men	Men	GOLD	GameTeam	Soviet Union team	

4	Equestrian Jumping	seoul-1988	team mixed		Open	BRONZE	GameTeam	France team

In [58]:
```
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 [59]:
```
olap_connection.commit()
```

In [60]:
```
olap_olympic_df.to_sql("olap_olympic_medals", con=olap_engine, if_exists="append", index
```

Out[60]:
```
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 [61]:
```
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 [62]:
```
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;
```

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 [63]:
```
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 [64]:
```
olap_cursor.execute("""
ALTER TABLE olap_olympic_medals
ADD COLUMN year INTEGER;
""")
```

In [65]:
```
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 [66]:
```
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 [67]:
```
fact_olympic_measure_data_columns = [desc[0] for desc in olap_cursor.description]
fact_olympic_measure_data_columns
```

Out[67]:
```
['country_code',
 'participant_id',
 'athlete_id',
 'event_id',
 'game_slug',
 'year',
 'total_bronze_medals',
 'total_silver_medals',
 'total_gold_medals',
 'total_medals']
```

In [68]:
```
fact_olympic_measure_df =  pd.DataFrame(fact_olympic_measure_data, columns = fact_olympi
fact_olympic_measure_df
```

Out[68]:

	country_code	participant_id	athlete_id	event_id	game_slug	year	total_bronze_medals	total_silver_me
0	GRC	NaN	NaN	129	athens-1896	1896	0	
1	GRC	NaN	NaN	455	athens-1896	1896	0	
2	GRC	NaN	NaN	611	athens-	1896	2	

					1896		
3	USA	NaN	NaN	694	athens-1896	1896	0
4	USA	NaN	NaN	873	athens-1896	1896	1
...	...	...	...	...	...	...	...
21578	TWN	NaN	342.0	1101	beijing-2022	2022	1
21579	SVN	NaN	11017.0	1546	beijing-2022	2022	0
21580	TWN	NaN	318.0	935	beijing-2022	2022	0
21581	AUT	NaN	305.0	1095	beijing-2022	2022	0
21582	NOR	NaN	5466.0	724	beijing-2022	2022	0

21583 rows × 10 columns

In [69]:
```
fact_olympic_measure_df.to_sql("factolympicmedalsmeasures", con=olap_engine, if_exists="
```

Out[69]:
```
583
```

Economic Measure

In [70]:
```
oltp_cursor.execute("""
SELECT time_year as year,
    country_code,
    poverty_ratio as poverty_count,
    gdp_per_capita_usd as gdp_per_capita,
    gdp_per_capita_growth as annual_gdp_growth,
    secure_internet_servers_per_million as servers_count
FROM economic_data;
""")

fact_economic_measures = oltp_cursor.fetchall()
```

In [71]:
```
oltp_connection.commit()
```

In [72]:
```
fact_economic_measures_columns = [desc[0] for desc in oltp_cursor.description]
fact_economic_measures_columns
```

Out[72]:
```
['year',
 'country_code',
 'poverty_count',
 'gdp_per_capita',
 'annual_gdp_growth',
 'servers_count']
```

In [73]:
```
fact_economic_measures_df =  pd.DataFrame(fact_economic_measures, columns = fact_economi
fact_economic_measures_df
```

Out[73]:

	year	country_code	poverty_count	gdp_per_capita	annual_gdp_growth	servers_count
0	1973	DEU	0.0	5046.755103	4.448017	1049.317943
1	2006	TUV	3.6	2402.480106	1.233859	92.038656
2	1960	ABW	8.2	6283.001443	16.263941	89.694143

3	1961	ABW	8.2	6283.001443	16.263941	89.694143
4	1962	ABW	8.2	6283.001443	16.263941	89.694143
...	...	...	...	...	...	...
13819	2019	ZWE	39.8	1421.868596	-8.177320	64.540886
13820	2020	ZWE	39.8	1372.696674	-9.670405	70.646050
13821	2021	ZWE	39.8	1773.920411	6.271613	70.646050
13822	2022	ZWE	39.8	1676.821489	4.387997	70.646050
13823	2023	ZWE	39.8	1676.821489	4.387997	70.646050

13824 rows × 6 columns

In [74]:
```
fact_economic_measures_df.to_sql("facteconomicmeasure", con=olap_engine, if_exists="appe
```

Out[74]:
```
824
```

In [75]:
```
# Extract health measure from the existing tables
oltp_cursor.execute("""
SELECT
    COALESCE(e.time_year, CAST(l.year AS VARCHAR(255)), CAST(m.year AS VARCHAR(255))) AS
    COALESCE(e.country_code, l.country_code, m.country_code) AS country_code,
    m.daly_depression,
    m.daly_schizophrenia,
    m.daly_bipolar_disorder,
    m.daly_eating_disorder,
    m.daly_anxiety,
    l.life_expectancy,
    e.infant_mortality_rate,
    e.health_expenditure_pct_gdp AS current_health_expenditure,
    e.gov_health_expenditure_per_capita_usd AS government_health_expenditure,
    e.private_health_expenditure_per_capita_usd AS private_health_expenditure,
    e.external_health_expenditure_per_capita_usd AS external_health_expenditure
FROM
    economic_data e
FULL OUTER JOIN
    life_expectancy_data l ON e.time_year = CAST(l.year AS VARCHAR(255)) AND e.country_c
FULL OUTER JOIN
    mental_health_data m ON COALESCE(e.time_year, CAST(l.year AS VARCHAR(255))) = CAST(m
""")
```

In [76]:
```
olap_health_measure = oltp_cursor.fetchall()
```

In [77]:
```
olap_health_measure_columns = [desc[0] for desc in oltp_cursor.description]
olap_health_measure_columns
```

Out[77]:
```
['year',
 'country_code',
 'daly_depression',
 'daly_schizophrenia',
 'daly_bipolar_disorder',
 'daly_eating_disorder',
 'daly_anxiety',
 'life_expectancy',
 'infant_mortality_rate',
 'current_health_expenditure',
 'government_health_expenditure',
 'private_health_expenditure',
 'external_health_expenditure']
```

In [78]:
```
olap_health_measure_df =  pd.DataFrame(olap_health_measure, columns = olap_health_measur
olap_health_measure_df
```

Out[78]:

	year	country_code	daly_depression	daly_schizophrenia	daly_bipolar_disorder	daly_eating_disorder	da
0	1543	GBR	NaN	NaN	NaN	NaN	
1	1548	GBR	NaN	NaN	NaN	NaN	
2	1553	GBR	NaN	NaN	NaN	NaN	
3	1558	GBR	NaN	NaN	NaN	NaN	
4	1563	GBR	NaN	NaN	NaN	NaN	
...	...	...	...	...	...	...	
19399	2023	WSM	NaN	NaN	NaN	NaN	
19400	2023	YEM	NaN	NaN	NaN	NaN	
19401	2023	ZAF	NaN	NaN	NaN	NaN	
19402	2023	ZMB	NaN	NaN	NaN	NaN	
19403	2023	ZWE	NaN	NaN	NaN	NaN	

19404 rows × 13 columns

In [79]:
```
olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').apply(


# Display the first few rows after grouped interpolation to verify changes
olap_health_measure_interpolated
```

```
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
```

```
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: FutureWarning: DataFrame.interpolate with object d
type is deprecated and will raise in a future version. Call obj.infer_objects(copy=Fals
e) before interpolating instead.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
/tmp/ipykernel_93818/2300131677.py:1: DeprecationWarning: DataFrameGroupBy.apply operate
d on the grouping columns. This behavior is deprecated, and in a future version of panda
s the grouping columns will be excluded from the operation. Either pass `include_groups=
False` to exclude the groupings or explicitly select the grouping columns after groupby
 to silence this warning.
  olap_health_measure_interpolated = olap_health_measure_df.groupby('country_code').appl
y(lambda group: group.interpolate(method='linear', limit_direction='both'))
```

Out[79]:

		year	country_code	daly_depression	daly_schizophrenia	daly_bipolar_disorder	daly_eati
country_code							
ABW	**1990**	1950	ABW	NaN	NaN	NaN	
	2225	1951	ABW	NaN	NaN	NaN	
	2460	1952	ABW	NaN	NaN	NaN	
	2695	1953	ABW	NaN	NaN	NaN	
	2930	1954	ABW	NaN	NaN	NaN	
...	**...**	...	...	...	...	...	
ZWE	**18499**	2019	ZWE	544.5316	128.21603	115.38145	
	18735	2020	ZWE	544.5316	128.21603	115.38145	
	18971	2021	ZWE	544.5316	128.21603	115.38145	
	19187	2022	ZWE	544.5316	128.21603	115.38145	
	19403	2023	ZWE	544.5316	128.21603	115.38145	

19404 rows × 13 columns

In [80]:

```python
# Calculate the median for each column and fill missing values with the median
columns_with_na = olap_health_measure_interpolated.columns[olap_health_measure_interpola
medians = olap_health_measure_interpolated[columns_with_na].median()

# Fill missing values with medians
olap_health_measure_interpolated_filled = olap_health_measure_interpolated.fillna(median

# Check the summary of missing values after filling to confirm
olap_health_measure_interpolated_filled.isna().sum()
```

Out[80]:

```
year                              0
country_code                      0
daly_depression                   0
daly_schizophrenia                0
```

```
daly_bipolar_disorder            0
daly_eating_disorder             0
daly_anxiety                     0
life_expectancy                  0
infant_mortality_rate            0
current_health_expenditure       0
government_health_expenditure    0
private_health_expenditure       0
external_health_expenditure      0
dtype: int64
```

In [81]:
```
olap_health_measure_interpolated_filled
```

Out[81]:

		year	country_code	daly_depression	daly_schizophrenia	daly_bipolar_disorder	daly_eati
country_code							
ABW	**1990**	1950	ABW	652.26294	175.044965	131.12288	
	2225	1951	ABW	652.26294	175.044965	131.12288	
	2460	1952	ABW	652.26294	175.044965	131.12288	
	2695	1953	ABW	652.26294	175.044965	131.12288	
	2930	1954	ABW	652.26294	175.044965	131.12288	
...	**...**	...	...	...	...	...	
ZWE	**18499**	2019	ZWE	544.53160	128.216030	115.38145	
	18735	2020	ZWE	544.53160	128.216030	115.38145	
	18971	2021	ZWE	544.53160	128.216030	115.38145	
	19187	2022	ZWE	544.53160	128.216030	115.38145	
	19403	2023	ZWE	544.53160	128.216030	115.38145	

19404 rows × 13 columns

In [82]:
```
olap_health_measure_interpolated_filled.to_sql("facthealthmeasure", con=olap_engine, if_
```

Out[82]:
```
404
```

In [83]:
```
# Extract relevant data for combined fact table

olap_cursor.execute("""
SELECT
    olympics.game_slug,
    olympics.participant_id,
    olympics.athlete_id,
    olympics.event_id,
    COALESCE(olympics.country_code, econ.country_code, health.country_code) AS country_c
    COALESCE(olympics.year, econ.year, health.year) AS year,
    olympics.total_bronze_medals,
    olympics.total_silver_medals,
    olympics.total_gold_medals,
    olympics.total_medals,
    econ.poverty_count,
    econ.gdp_per_capita,
    econ.annual_gdp_growth,
    econ.servers_count,
    health.daly_depression,
    health.daly_schizophrenia,
    health.daly_bipolar_disorder,
    health.daly_eating_disorder,
    health.daly_anxiety,
```

```
    health.life_expectancy,
    health.infant_mortality_rate,
    health.current_health_expenditure,
    health.government_health_expenditure,
    health.private_health_expenditure,
    health.external_health_expenditure
FROM FactOlympicMedalsMeasures AS olympics
FULL OUTER JOIN FactEconomicMeasure AS econ
    ON olympics.country_code = econ.country_code AND olympics.year = econ.year
FULL OUTER JOIN FactHealthMeasure AS health
    ON COALESCE(olympics.country_code, econ.country_code) = health.country_code
    AND COALESCE(olympics.year, econ.year) = health.year;
""")
```

In [84]:
```
combined_fact_table = olap_cursor.fetchall()
```

In [85]:
```
combine_fact_table_columns = [desc[0] for desc in olap_cursor.description]
combine_fact_table_columns
```

Out[85]:
```
['game_slug',
 'participant_id',
 'athlete_id',
 'event_id',
 'country_code',
 'year',
 'total_bronze_medals',
 'total_silver_medals',
 'total_gold_medals',
 'total_medals',
 'poverty_count',
 'gdp_per_capita',
 'annual_gdp_growth',
 'servers_count',
 'daly_depression',
 'daly_schizophrenia',
 'daly_bipolar_disorder',
 'daly_eating_disorder',
 'daly_anxiety',
 'life_expectancy',
 'infant_mortality_rate',
 'current_health_expenditure',
 'government_health_expenditure',
 'private_health_expenditure',
 'external_health_expenditure']
```

In [86]:
```
combined_fact_table_df =  pd.DataFrame(combined_fact_table, columns = combine_fact_table
combined_fact_table_df
```

Out[86]:

	game_slug	participant_id	athlete_id	event_id	country_code	year	total_bronze_medals	total_silver_me
0	None	NaN	NaN	NaN	ABW	1950	NaN	
1	None	NaN	NaN	NaN	ABW	1951	NaN	
2	None	NaN	NaN	NaN	ABW	1952	NaN	
3	None	NaN	NaN	NaN	ABW	1953	NaN	
4	None	NaN	NaN	NaN	ABW	1954	NaN	
...	...	...	...	...	...	...	...	
39693	None	NaN	NaN	NaN	ZWE	2019	NaN	
39694	None	NaN	NaN	NaN	ZWE	2020	NaN	
39695	None	NaN	NaN	NaN	ZWE	2021	NaN	
39696	None	NaN	NaN	NaN	ZWE	2022	NaN	

39697	None	NaN	NaN	NaN	ZWE	2023		NaN

39698 rows × 25 columns

In [88]:
```
combined_fact_table_df.to_sql("combinedfacttable", con=olap_engine, if_exists="append",
```

Out[88]:
```
698
```

In [ ]:
```

```

Cube and Data Warehouse

In [1]:
```
import atoti as tt
```

```
Welcome to Atoti 0.8.10!

By using this community edition, you agree with the license available at https://docs.at
oti.io/latest/eula.html.
Browse the official documentation at https://docs.atoti.io.
Join the community at https://www.atoti.io/register.

Atoti collects telemetry data, which is used to help understand how to improve the produ
ct.
If you don't wish to send usage data, you can request a trial license at https://www.ato
ti.io/evaluation-license-request.

You can hide this message by setting the `ATOTI_HIDE_EULA_MESSAGE` environment variable
to True.
```

In [2]:
```
session = tt.Session(
    user_content_storage=".content",
    port=9092,
    java_options=["-Xms1G", "-Xmx10G"]
)
```

In [3]:
```
db_name = "olympic_olap"
db_user = "postgres"
db_password = "postgres"
db_host = "pgdb"  # Update if your DB is hosted elsewhere
db_port = "5432"

jdbc_url = f"jdbc:postgresql://{db_host}:{db_port}/{db_name}?user={db_user}&password={db
```

In [4]:
```
# # Load the combined fact table from the database into the Atoti session.
# This table aggregates various metrics across multiple dimensions including sports, eco
# and health data for comprehensive analysis.

combined_fact_table = session.read_sql(
    """
    SELECT
        id,
        game_slug,
        participant_id,
        athlete_id,
        event_id,
        country_code,
        CAST(year AS CHAR(4)) AS year,  -- Casting year to string directly in SQL
        total_bronze_medals,
        total_silver_medals,
        total_gold_medals,
        total_medals,
        poverty_count,
        gdp_per_capita,
        annual_gdp_growth,
        servers_count,
        daly_depression,
        daly_schizophrenia,
        daly_bipolar_disorder,
        daly_eating_disorder,
        daly_anxiety,
        life_expectancy,
```

```
        infant_mortality_rate,
        current_health_expenditure,
        government_health_expenditure,
        private_health_expenditure,
        external_health_expenditure
    FROM combinedfacttable
    """,
    keys=["id"],
    table_name="CombinedFactTable",
    url=jdbc_url,
)
```

In [5]:
```
# # Load the dimension table for location data from the database into the Atoti session.
# This table includes geographic details indexed by country code to enhance analytical c

dimlocation_table = session.read_sql(
    "SELECT * FROM DimLocation",
    keys=["country_code"],
    table_name="DimLocation",
    url=jdbc_url)
```

In [6]:
```
# Load the dimension table for event data from the database into the Atoti session.

dimevent_table = session.read_sql(
    "SELECT * FROM DimEvent",
    keys=["event_id"],
    table_name="DimEvent",
    url=jdbc_url,
)
```

In [7]:
```
# Load the dimension table for participant data from the database into the Atoti session

dimparticipant_table = session.read_sql(
    "SELECT * FROM DimParticipant",
    keys=["participant_id"],
    table_name="DimParticipant",
    url=jdbc_url,
)
```

In [8]:
```
# Load the dimension table for athlete data from the database into the Atoti session.

dimathlete_table = session.read_sql(
    "SELECT * FROM DimAthlete",
    keys=["athlete_id"],
    table_name="DimAthlete",
    url=jdbc_url,
)
```

In [9]:
```
# Load the dimension table for year data from the database into the Atoti session.

dimyear_table = session.read_sql(
    "SELECT CAST(year AS CHAR(4)) AS year FROM DimYear",
    keys=["year"],
    table_name="DimYear",
    url=jdbc_url,
)
```

In [10]:
```
# Load the dimension table for game data from the database into the Atoti session.

dimgame_table = session.read_sql(
    "SELECT * FROM DimGame",
    keys=["game_slug"],
    table_name="DimGame",
```

```
    url=jdbc_url,
)
```

In [11]:
```
# Join the combined fact table with various dimension tables to enrich the dataset with
# This will facilitate more detailed and comprehensive analysis across multiple dimensio

combined_fact_table.join(dimlocation_table, combined_fact_table["country_code"] == dimlo

combined_fact_table.join(dimevent_table, combined_fact_table["event_id"] == dimevent_tab

combined_fact_table.join(dimparticipant_table, combined_fact_table["participant_id"] ==

combined_fact_table.join(dimathlete_table, combined_fact_table["athlete_id"] == dimathle

combined_fact_table.join(dimyear_table, combined_fact_table["year"] == dimyear_table["ye

combined_fact_table.join(dimgame_table, combined_fact_table["game_slug"] == dimgame_tabl
```

In [12]:
```
combined_fact_table['year']
```

Out[12]:

- year
  - key: False
  - type: String
  - default_value: N/A

In [13]:
```
session.tables.schema
```

Out[13]:
```
mermaid
erDiagram
  "DimAthlete" {
    _ int PK "athlete_id"
    _ String "athlete_name"
    _ String "athlete_url"
  }
  "DimLocation" {
    _ String PK "country_code"
    _ String "country_name"
    _ String "continent"
  }
  "DimEvent" {
    _ int PK "event_id"
    _ String "event_title"
    _ String "event_discipline"
    _ String "event_gender"
  }
  "CombinedFactTable" {
    _ int PK "id"
    _ String "game_slug"
    nullable int "participant_id"
    nullable int "athlete_id"
    nullable int "event_id"
    _ String "country_code"
    _ String "year"
    nullable int "total_bronze_medals"
    nullable int "total_silver_medals"
    nullable int "total_gold_medals"
    nullable int "total_medals"
    nullable double "poverty_count"
```

```
    nullable double "gdp_per_capita"
    nullable double "annual_gdp_growth"
    nullable int "servers_count"
    nullable double "daly_depression"
    nullable double "daly_schizophrenia"
    nullable double "daly_bipolar_disorder"
    nullable double "daly_eating_disorder"
    nullable double "daly_anxiety"
    nullable double "life_expectancy"
    nullable double "infant_mortality_rate"
    nullable double "current_health_expenditure"
    nullable double "government_health_expenditure"
    nullable double "private_health_expenditure"
    nullable double "external_health_expenditure"
  }
  "DimParticipant" {
    _ int PK "participant_id"
    _ String "participant_title"
    _ String "participant_type"
  }
  "DimYear" {
    _ String PK "year"
  }
  "DimGame" {
    _ String PK "game_slug"
    _ String "game_name"
    _ String "game_season"
    nullable int "game_year"
    _ String "country_code"
  }
  "CombinedFactTable" }o--o| "DimLocation" : "`country_code` == `country_code`"
  "CombinedFactTable" }o--o| "DimGame" : "`game_slug` == `game_slug`"
  "CombinedFactTable" }o--o| "DimParticipant" : "`participant_id` == `participan
t_id`"
  "CombinedFactTable" }o--o| "DimYear" : "`year` == `year`"
  "CombinedFactTable" }o--o| "DimAthlete" : "`athlete_id` == `athlete_id`"
  "CombinedFactTable" }o--o| "DimEvent" : "`event_id` == `event_id`"
```

In [14]:
```
# Create a cube from the combined fact table in the Atoti session.
# This cube will allow for multi-dimensional analysis and visualization of the data aggr
# facilitating deeper insights and easier exploration.

cube = session.create_cube(combined_fact_table)
```

In [15]:
```
cube
```

Out[15]:

- CombinedFactTable
  - Dimensions
    - CombinedFactTable
      - country_code
        1. country_code
      - game_slug
        1. game_slug
      - id
        1. id
      - year

            1. year
    - DimAthlete
        - athlete_name
            1. athlete_name
        - athlete_url
            1. athlete_url
    - DimEvent
        - event_discipline
            1. event_discipline
        - event_gender
            1. event_gender
        - event_title
            1. event_title
    - DimGame
        - country_code
            1. country_code
        - game_name
            1. game_name
        - game_season
            1. game_season
    - DimLocation
        - continent
            1. continent
        - country_name
            1. country_name
    - DimParticipant
        - participant_title
            1. participant_title
        - participant_type
            1. participant_type
- Measures
    - annual_gdp_growth.MEAN
        - formatter: DOUBLE[#,###.00]
    - annual_gdp_growth.SUM
        - formatter: DOUBLE[#,###.00]
    - athlete_id.MEAN
        - formatter: DOUBLE[#,###.00]
    - athlete_id.SUM
        - formatter: INT[#,###]
    - contributors.COUNT
        - formatter: INT[#,###]
    - current_health_expenditure.MEAN
        - formatter: DOUBLE[#,###.00]
    - current_health_expenditure.SUM
        - formatter: DOUBLE[#,###.00]
    - daly_anxiety.MEAN
        - formatter: DOUBLE[#,###.00]
    - daly_anxiety.SUM

    - formatter: DOUBLE[#,###.00]
- daly_bipolar_disorder.MEAN
    - formatter: DOUBLE[#,###.00]
- daly_bipolar_disorder.SUM
    - formatter: DOUBLE[#,###.00]
- daly_depression.MEAN
    - formatter: DOUBLE[#,###.00]
- daly_depression.SUM
    - formatter: DOUBLE[#,###.00]
- daly_eating_disorder.MEAN
    - formatter: DOUBLE[#,###.00]
- daly_eating_disorder.SUM
    - formatter: DOUBLE[#,###.00]
- daly_schizophrenia.MEAN
    - formatter: DOUBLE[#,###.00]
- daly_schizophrenia.SUM
    - formatter: DOUBLE[#,###.00]
- event_id.MEAN
    - formatter: DOUBLE[#,###.00]
- event_id.SUM
    - formatter: INT[#,###]
- external_health_expenditure.MEAN
    - formatter: DOUBLE[#,###.00]
- external_health_expenditure.SUM
    - formatter: DOUBLE[#,###.00]
- gdp_per_capita.MEAN
    - formatter: DOUBLE[#,###.00]
- gdp_per_capita.SUM
    - formatter: DOUBLE[#,###.00]
- government_health_expenditure.MEAN
    - formatter: DOUBLE[#,###.00]
- government_health_expenditure.SUM
    - formatter: DOUBLE[#,###.00]
- infant_mortality_rate.MEAN
    - formatter: DOUBLE[#,###.00]
- infant_mortality_rate.SUM
    - formatter: DOUBLE[#,###.00]
- life_expectancy.MEAN
    - formatter: DOUBLE[#,###.00]
- life_expectancy.SUM
    - formatter: DOUBLE[#,###.00]
- participant_id.MEAN
    - formatter: DOUBLE[#,###.00]
- participant_id.SUM
    - formatter: INT[#,###]
- poverty_count.MEAN
    - formatter: DOUBLE[#,###.00]
- poverty_count.SUM

        - formatter: DOUBLE[#,###.00]
    - private_health_expenditure.MEAN
        - formatter: DOUBLE[#,###.00]
    - private_health_expenditure.SUM
        - formatter: DOUBLE[#,###.00]
    - servers_count.MEAN
        - formatter: DOUBLE[#,###.00]
    - servers_count.SUM
        - formatter: INT[#,###]
    - total_bronze_medals.MEAN
        - formatter: DOUBLE[#,###.00]
    - total_bronze_medals.SUM
        - formatter: INT[#,###]
    - total_gold_medals.MEAN
        - formatter: DOUBLE[#,###.00]
    - total_gold_medals.SUM
        - formatter: INT[#,###]
    - total_medals.MEAN
        - formatter: DOUBLE[#,###.00]
    - total_medals.SUM
        - formatter: INT[#,###]
    - total_silver_medals.MEAN
        - formatter: DOUBLE[#,###.00]
    - total_silver_medals.SUM
        - formatter: INT[#,###]

In [16]:
```
hierarchies, levels, measures = cube.hierarchies, cube.levels, cube.measures
```

In [17]:
```
hierarchies
```

Out[17]:

- Dimensions
    - CombinedFactTable
        - country_code
            1. country_code
        - game_slug
            1. game_slug
        - id
            1. id
        - year
            1. year
    - DimAthlete
        - athlete_name
            1. athlete_name
        - athlete_url
            1. athlete_url
    - DimEvent
        - event_discipline
            1. event_discipline

        - event_gender
            1. event_gender
        - event_title
            1. event_title
    - DimGame
        - country_code
            1. country_code
        - game_name
            1. game_name
        - game_season
            1. game_season
    - DimLocation
        - continent
            1. continent
        - country_name
            1. country_name
    - DimParticipant
        - participant_title
            1. participant_title
        - participant_type
            1. participant_type

In [18]:
```
hierarchies["Year"] = [dimyear_table["year"]]
```

In [19]:
```
hierarchies
```

Out[19]:

- Dimensions
    - CombinedFactTable
        - country_code
            1. country_code
        - game_slug
            1. game_slug
        - id
            1. id
        - year
            1. year
    - DimAthlete
        - athlete_name
            1. athlete_name
        - athlete_url
            1. athlete_url
    - DimEvent
        - event_discipline
            1. event_discipline
        - event_gender
            1. event_gender
        - event_title
            1. event_title

- DimGame
  - country_code
    1. country_code
  - game_name
    1. game_name
  - game_season
    1. game_season
- DimLocation
  - continent
    1. continent
  - country_name
    1. country_name
- DimParticipant
  - participant_title
    1. participant_title
  - participant_type
    1. participant_type
- DimYear
  - Year
    1. year

In [20]:
```
# List all hierarchies in the cube to find the correct name
for hierarchy in hierarchies:
    print(hierarchy)
```

```
('CombinedFactTable', 'country_code')
('CombinedFactTable', 'id')
('DimParticipant', 'participant_title')
('DimAthlete', 'athlete_name')
('DimGame', 'game_season')
('CombinedFactTable', 'game_slug')
('DimParticipant', 'participant_type')
('DimEvent', 'event_discipline')
('CombinedFactTable', 'year')
('DimEvent', 'event_gender')
('DimYear', 'Year')
('DimLocation', 'country_name')
('DimAthlete', 'athlete_url')
('DimEvent', 'event_title')
('DimGame', 'game_name')
('DimLocation', 'continent')
('DimGame', 'country_code')
```

In [21]:
```
# Remove unwanted hierarchy
del hierarchies[('CombinedFactTable', 'country_code')]
del hierarchies[('CombinedFactTable', 'id')]
del hierarchies[('CombinedFactTable', 'game_slug')]
del hierarchies[('CombinedFactTable', 'year')]
```

In [22]:
```
hierarchies
```

Out[22]:

- Dimensions
  - DimAthlete
    - athlete_name
      1. athlete_name

        - athlete_url
            1. athlete_url
    - DimEvent
        - event_discipline
            1. event_discipline
        - event_gender
            1. event_gender
        - event_title
            1. event_title
    - DimGame
        - country_code
            1. country_code
        - game_name
            1. game_name
        - game_season
            1. game_season
    - DimLocation
        - continent
            1. continent
        - country_name
            1. country_name
    - DimParticipant
        - participant_title
            1. participant_title
        - participant_type
            1. participant_type
    - DimYear
        - Year
            1. year

In [23]:
```
levels
```

Out[23]:

- Levels
    - athlete_name (DimAthlete/athlete_name/athlete_name)
        - dimension: DimAthlete
        - hierarchy: athlete_name
        - type: String
        - order: NaturalOrder
    - athlete_url (DimAthlete/athlete_url/athlete_url)
        - dimension: DimAthlete
        - hierarchy: athlete_url
        - type: String
        - order: NaturalOrder
    - continent (DimLocation/continent/continent)
        - dimension: DimLocation
        - hierarchy: continent
        - type: String
        - order: NaturalOrder

- country_code (DimGame/country_code/country_code)
  - dimension: DimGame
  - hierarchy: country_code
  - type: String
  - order: NaturalOrder
- country_name (DimLocation/country_name/country_name)
  - dimension: DimLocation
  - hierarchy: country_name
  - type: String
  - order: NaturalOrder
- event_discipline (DimEvent/event_discipline/event_discipline)
  - dimension: DimEvent
  - hierarchy: event_discipline
  - type: String
  - order: NaturalOrder
- event_gender (DimEvent/event_gender/event_gender)
  - dimension: DimEvent
  - hierarchy: event_gender
  - type: String
  - order: NaturalOrder
- event_title (DimEvent/event_title/event_title)
  - dimension: DimEvent
  - hierarchy: event_title
  - type: String
  - order: NaturalOrder
- game_name (DimGame/game_name/game_name)
  - dimension: DimGame
  - hierarchy: game_name
  - type: String
  - order: NaturalOrder
- game_season (DimGame/game_season/game_season)
  - dimension: DimGame
  - hierarchy: game_season
  - type: String
  - order: NaturalOrder
- participant_title (DimParticipant/participant_title/participant_title)
  - dimension: DimParticipant
  - hierarchy: participant_title
  - type: String
  - order: NaturalOrder
- participant_type (DimParticipant/participant_type/participant_type)
  - dimension: DimParticipant
  - hierarchy: participant_type
  - type: String
  - order: NaturalOrder
- year (DimYear/Year/year)
  - dimension: DimYear
  - hierarchy: Year

- type: String
- order: NaturalOrder

In [24]:
```
# Define new hierarchies`

hierarchies["DimAthlete"] = [levels["athlete_name"]]

hierarchies["DimEvent"] = [levels["event_title"], levels[("DimEvent", "event_discipline"
                           levels[("DimEvent", "event_gender", "event_gender")]]

hierarchies["DimGame"] = [levels["game_name"], levels[("DimGame", "game_season", "game_s

hierarchies["DimLocation"]  = [levels["country_name"], levels[("DimLocation", "continent

hierarchies["DimParticipant"] = [levels["participant_title"], levels[("DimParticipant",

hierarchies["DimYear"] = [levels["year"]]
```

In [25]:
```
# Delete unnecessary hierarchy

del hierarchies[("DimAthlete", "athlete_name")]

del hierarchies[('DimAthlete', 'athlete_url')]

del hierarchies[('DimEvent', 'event_gender')]
del hierarchies[('DimEvent', 'event_title')]
del hierarchies[('DimEvent', 'event_discipline')]

del hierarchies[('DimGame', 'game_season')]
del hierarchies[('DimGame', 'country_code')]
del hierarchies[('DimGame', 'game_name')]

del hierarchies[('DimLocation', 'country_name')]
del hierarchies[('DimLocation', 'continent')]


del hierarchies[('DimParticipant', 'participant_title')]
del hierarchies[('DimParticipant', 'participant_type')]

del hierarchies[('DimYear', 'Year')]
```

In [26]:
```
hierarchies
```

Out[26]:

- Dimensions
  - DimAthlete
    - DimAthlete
      1. athlete_name
  - DimEvent
    - DimEvent
      1. event_title
      2. event_discipline
      3. event_gender
  - DimGame
    - DimGame
      1. game_name
      2. game_season
  - DimLocation

        - DimLocation
            1. country_name
            2. continent
    - DimParticipant
        - DimParticipant
            1. participant_title
            2. participant_type
    - DimYear
        - DimYear
            1. year

In [27]:
```
measures
```

Out[27]:

- Measures
    - annual_gdp_growth.MEAN
        - formatter: DOUBLE[#,###.00]
    - annual_gdp_growth.SUM
        - formatter: DOUBLE[#,###.00]
    - athlete_id.MEAN
        - formatter: DOUBLE[#,###.00]
    - athlete_id.SUM
        - formatter: INT[#,###]
    - contributors.COUNT
        - formatter: INT[#,###]
    - current_health_expenditure.MEAN
        - formatter: DOUBLE[#,###.00]
    - current_health_expenditure.SUM
        - formatter: DOUBLE[#,###.00]
    - daly_anxiety.MEAN
        - formatter: DOUBLE[#,###.00]
    - daly_anxiety.SUM
        - formatter: DOUBLE[#,###.00]
    - daly_bipolar_disorder.MEAN
        - formatter: DOUBLE[#,###.00]
    - daly_bipolar_disorder.SUM
        - formatter: DOUBLE[#,###.00]
    - daly_depression.MEAN
        - formatter: DOUBLE[#,###.00]
    - daly_depression.SUM
        - formatter: DOUBLE[#,###.00]
    - daly_eating_disorder.MEAN
        - formatter: DOUBLE[#,###.00]
    - daly_eating_disorder.SUM
        - formatter: DOUBLE[#,###.00]
    - daly_schizophrenia.MEAN
        - formatter: DOUBLE[#,###.00]
    - daly_schizophrenia.SUM
        - formatter: DOUBLE[#,###.00]

- event_id.MEAN
  - formatter: DOUBLE[#,###.00]
- event_id.SUM
  - formatter: INT[#,###]
- external_health_expenditure.MEAN
  - formatter: DOUBLE[#,###.00]
- external_health_expenditure.SUM
  - formatter: DOUBLE[#,###.00]
- gdp_per_capita.MEAN
  - formatter: DOUBLE[#,###.00]
- gdp_per_capita.SUM
  - formatter: DOUBLE[#,###.00]
- government_health_expenditure.MEAN
  - formatter: DOUBLE[#,###.00]
- government_health_expenditure.SUM
  - formatter: DOUBLE[#,###.00]
- infant_mortality_rate.MEAN
  - formatter: DOUBLE[#,###.00]
- infant_mortality_rate.SUM
  - formatter: DOUBLE[#,###.00]
- life_expectancy.MEAN
  - formatter: DOUBLE[#,###.00]
- life_expectancy.SUM
  - formatter: DOUBLE[#,###.00]
- participant_id.MEAN
  - formatter: DOUBLE[#,###.00]
- participant_id.SUM
  - formatter: INT[#,###]
- poverty_count.MEAN
  - formatter: DOUBLE[#,###.00]
- poverty_count.SUM
  - formatter: DOUBLE[#,###.00]
- private_health_expenditure.MEAN
  - formatter: DOUBLE[#,###.00]
- private_health_expenditure.SUM
  - formatter: DOUBLE[#,###.00]
- servers_count.MEAN
  - formatter: DOUBLE[#,###.00]
- servers_count.SUM
  - formatter: INT[#,###]
- total_bronze_medals.MEAN
  - formatter: DOUBLE[#,###.00]
- total_bronze_medals.SUM
  - formatter: INT[#,###]
- total_gold_medals.MEAN
  - formatter: DOUBLE[#,###.00]
- total_gold_medals.SUM
  - formatter: INT[#,###]

- total_medals.MEAN
  - formatter: DOUBLE[#,###.00]
- total_medals.SUM
  - formatter: INT[#,###]
- total_silver_medals.MEAN
  - formatter: DOUBLE[#,###.00]
- total_silver_medals.SUM
  - formatter: INT[#,###]

In [28]:
```
del measures["athlete_id.MEAN"]


del measures["contributors.COUNT"]

del measures["athlete_id.SUM"]

del measures["event_id.MEAN"]

del measures["event_id.SUM"]
del measures["participant_id.MEAN"]
del measures["participant_id.SUM"]
```

In [29]:
```
measures
```

Out[29]:

- Measures
  - annual_gdp_growth.MEAN
    - formatter: DOUBLE[#,###.00]
  - annual_gdp_growth.SUM
    - formatter: DOUBLE[#,###.00]
  - current_health_expenditure.MEAN
    - formatter: DOUBLE[#,###.00]
  - current_health_expenditure.SUM
    - formatter: DOUBLE[#,###.00]
  - daly_anxiety.MEAN
    - formatter: DOUBLE[#,###.00]
  - daly_anxiety.SUM
    - formatter: DOUBLE[#,###.00]
  - daly_bipolar_disorder.MEAN
    - formatter: DOUBLE[#,###.00]
  - daly_bipolar_disorder.SUM
    - formatter: DOUBLE[#,###.00]
  - daly_depression.MEAN
    - formatter: DOUBLE[#,###.00]
  - daly_depression.SUM
    - formatter: DOUBLE[#,###.00]
  - daly_eating_disorder.MEAN
    - formatter: DOUBLE[#,###.00]
  - daly_eating_disorder.SUM
    - formatter: DOUBLE[#,###.00]
  - daly_schizophrenia.MEAN
    - formatter: DOUBLE[#,###.00]

- daly_schizophrenia.SUM
  - formatter: DOUBLE[#,###.00]
- external_health_expenditure.MEAN
  - formatter: DOUBLE[#,###.00]
- external_health_expenditure.SUM
  - formatter: DOUBLE[#,###.00]
- gdp_per_capita.MEAN
  - formatter: DOUBLE[#,###.00]
- gdp_per_capita.SUM
  - formatter: DOUBLE[#,###.00]
- government_health_expenditure.MEAN
  - formatter: DOUBLE[#,###.00]
- government_health_expenditure.SUM
  - formatter: DOUBLE[#,###.00]
- infant_mortality_rate.MEAN
  - formatter: DOUBLE[#,###.00]
- infant_mortality_rate.SUM
  - formatter: DOUBLE[#,###.00]
- life_expectancy.MEAN
  - formatter: DOUBLE[#,###.00]
- life_expectancy.SUM
  - formatter: DOUBLE[#,###.00]
- poverty_count.MEAN
  - formatter: DOUBLE[#,###.00]
- poverty_count.SUM
  - formatter: DOUBLE[#,###.00]
- private_health_expenditure.MEAN
  - formatter: DOUBLE[#,###.00]
- private_health_expenditure.SUM
  - formatter: DOUBLE[#,###.00]
- servers_count.MEAN
  - formatter: DOUBLE[#,###.00]
- servers_count.SUM
  - formatter: INT[#,###]
- total_bronze_medals.MEAN
  - formatter: DOUBLE[#,###.00]
- total_bronze_medals.SUM
  - formatter: INT[#,###]
- total_gold_medals.MEAN
  - formatter: DOUBLE[#,###.00]
- total_gold_medals.SUM
  - formatter: INT[#,###]
- total_medals.MEAN
  - formatter: DOUBLE[#,###.00]
- total_medals.SUM
  - formatter: INT[#,###]
- total_silver_medals.MEAN
  - formatter: DOUBLE[#,###.00]

- total_silver_medals.SUM
  - formatter: INT[#,###]

In [30]:
```
session.widget
```


In [31]:
```
session.link
```

Out[31]: http://localhost:9092

Note: This is the session's local URL: it may not be reachable if Atoti is running on another machine.

In [32]:
```
# m["Price.VALUE"] = tt.agg.single_value(table["Price"])

# Create new measures for data visualisation. Since, we require data point for each year

measures["daly_anxiety.VALUE"] = tt.agg.single_value(combined_fact_table["daly_anxiety"]

measures["daly_bipolar_disorder.VALUE"] = tt.agg.single_value(combined_fact_table["daly_

measures["daly_depression.VALUE"] = tt.agg.single_value(combined_fact_table["daly_depres

measures["daly_eating_disorder.VALUE"] = tt.agg.single_value(combined_fact_table["daly_e

measures["daly_schizophrenia.VALUE"] = tt.agg.single_value(combined_fact_table["daly_sch

measures["life_expectancy.VALUE"] = tt.agg.single_value(combined_fact_table["life_expect

measures["current_health_expenditure.VALUE"] = tt.agg.single_value(combined_fact_table["

measures["external_health_expenditure.VALUE"] = tt.agg.single_value(combined_fact_table[

measures["government_health_expenditure.VALUE"] = tt.agg.single_value(combined_fact_tabl

measures["infant_mortality_rate.VALUE"] = tt.agg.single_value(combined_fact_table["infan

measures["private_health_expenditure.VALUE"] = tt.agg.single_value(combined_fact_table["

measures["annual_gdp_growth.VALUE"] = tt.agg.single_value(combined_fact_table["annual_gd

measures["poverty_count.VALUE"] = tt.agg.single_value(combined_fact_table["poverty_count
```

The above measure have been used for the visualisation in the report.

In [33]:
```
measures["servers_count.VALUE"] = tt.agg.single_value(combined_fact_table["servers_count
```