Database Analytics and Performance on Dataset of Global Temperature Changes (1970 – 2021)

Tomáš Staroň, Michal Kvet
University of Žilina
Žilina, Slovakia
Michal.Kvet@fri.uniza.sk

***Abstract*—The main objective of this work is to demonstrate how various database optimization tools can be used in climate change research to enhance data processing and analytical efficiency. This study analyses global temperature changes across all countries from 1970 to 2021, leveraging optimization techniques available in Oracle Database. By applying indexing, materialized views, and partitioning strategies, we significantly improve query performance, enabling faster and more efficient data analysis. Key findings include the identification of countries with the most significant temperature rises, particularly in Arctic regions, and the detection of extreme interannual variations. This research highlights the important role of database optimization in climate studies, offering a framework for more efficient data management and analysis.**

I. INTRODUCTION

The dataset that is analysed in this work consists of data across all countries from 1970 to 2021. It includes information about global surface temperatures based on different sources including weather stations, satellites and ocean buoys. The temperature is measured by the unit of degree Celsius. The temperature is presented by the change from the baseline temperature that was measured as an average in the years 1951 and 1980. If The temperature increased, then the index is positive and if the temperature decreased the index is negative.

Analysing this dataset can provide valuable insights into the ongoing issue of global warming. By examining temperature changes across different geographic locations, we can assess whether temperatures are indeed rising and how climate patterns are evolving over time.

II. DATA STRUCTURE AND PROCESSING

Data are free to access on the website Kaggle.com [1] – [6]. The structure of the dataset is simple. CSV file which includes different columns:

- *ObjectId* - identifies the row of the dataset,
- *Country Name* – identifies the country,
- *Unit* – unit of temperature measurements,
- *Change* – Indicates the type of temperature measurements, in this case the temperature was always the surface temperature,
- *Year* – columns representing the change of the temperature in comparison with the baseline temperature, columns are for years 1970 to 2021.

To process the data, we decided to use the tool Oracle SQL developer. This tool works with the Oracle database which has available performance and analytic tools that we need to use [6] [8].

To load the data into our Oracle database we firstly needed to get rid of the unnecessary data. In this case it was column Unit and Change, since they included only 1 unique value. For this step we can use any free online tool which can extract column from csv file.

	ID	COUNTRYNAME	YEAR1970	YEAR1971	YEAR1972	YEAR1973	YEAR1974
55	75	Germany	-0.831	0.034	-0.014	0.188	0.318
56	76	Ghana	0.323	-0.122	0.042	0.582	0.018
57	77	Gibraltar	-0.088	-0.851	-0.913	-0.195	-0.408
58	78	Greece	0.255	-0.235	-0.196	-0.374	-0.208
59	79	Greenland	-0.229	-0.789	-0.846	-0.992	0.3
60	80	Grenada	0.234	-0.306	-0.103	0.104	-0.409
61	81	Guadeloupe	0.105	-0.21	-0.106	0.199	-0.338
62	82	Guatemala	-0.115	-0.155	0.409	0.433	-0.311
63	83	Guiana, French	0.275	-0.339	-0.03	0.184	-0.322

Fig. 1. Preview of data in the input csv file after extracting the unimportant columns.

Inside Oracle SQL developer we can import the data directly from csv file. If the file is valid (contains structured columns and rows) it can automatically create the table with desired structure.

In any case, certain complications may arise while using this tool. While importing our data we had to face the problem with the values in temperature change values. There was an issue with the localization. Our database expected that number values with decimal points would include “decimal commas”, but our csv file included “decimal dots”. This issue can be solved by different approaches. The easiest way to handle this was to run this SQL query:

ALTER session SET nls_territory = 'UNITED KINGDOM';

After applying this, the import tool then was able to handle the situation and convert the string from the csv file into number values to store them in our table. In result our table includes 227 rows for all the countries (226 countries and 1 row for average world value) and each with 54 columns (*ObjectId*, *Country_Name* and 52 temperature columns). Some countries may be missing.

III. TABLE TRANSFORMATION TO NORMALIZED FORM

Since SQL is more efficient for operations on longs tables (so-called normalized format), we can use relational operator

UNPIVOT [8] [9] to transform a wide table into a normalized form. In order to do this we ran a script in the figure below.

```
CREATE TABLE temp_changes_unpivoted AS
SELECT CountryName,
       TO_NUMBER(SUBSTR(year_col, 5, 4)) AS year,
       temp_change
FROM temperatures
UNPIVOT (temp_change FOR year_col IN (
    year1970, year1971, year1972, year1973, year1974, year1975, year1976,
    year1977, year1978, year1979, year1980, year1981, year1982, year1983,
    year1984, year1985, year1986, year1987, year1988, year1989, year1990,
    year1991, year1992, year1993, year1994, year1995, year1996, year1997,
    year1998, year1999, year2000, year2001, year2002, year2003, year2004,
    year2005, year2006, year2007, year2008, year2009, year2010, year2011,
    year2012, year2013, year2014, year2015, year2016, year2017, year2018,
    year2019, year2020, year2021
));
```

Fig. 2. SQL query that normalized imported data by using UNPIVOT relational operator

The result normalized data preview is in the figure below. Now we can access all necessary data by better approach.

	COUNTRYNAME	YEAR	TEMP_CHANGE
3072	Guinea-Bissau	2019	1.642
3073	Guinea-Bissau	2020	1.917
3074	Guinea-Bissau	2021	1.912
3075	Guyana	1970	0.438
3076	Guyana	1971	-0.473

Fig. 3. Preview of normalized data

The created table consists of 10896 rows.

Result table "*temp_changes_unpivoted*" columns:

- *ID* – Number data type and primary key of the table, which identifies the row,
- *COUNTRYNAME* – VARCHAR2 data type, identifying country by its name,
- *YEAR* – Number (38, 0) data type, indicates to which year, the current row information, is dedicated,
- *TEMPCHANGE* – Number (38, 0) data type, the value by which the temperature changed in comparison with the baseline temperature.

IV. INDEXES

Indexes are essential for optimizing database performance, enabling faster data retrieval by reducing the number of scanned rows. Instead of searching the entire table, indexes act as structured "shortcuts" that guide queries directly to relevant data [7] – [11].

In large datasets like our global temperature database, frequent queries on specific countries and years can be slow without proper indexing. By implementing indexes, we significantly improve query efficiency, making data analysis more responsive and scalable [9].

A. Base index

When a primary key is defined on a table, Oracle Database automatically creates a unique index on the corresponding column(s). In our case, an index is created on the ID column, ensuring fast lookups and efficient access to individual rows. This functions as the default access path for retrieving data from the table.

B. Creating additional indexes

We have established a hypothesis that creating additional indexes on frequently queried columns will improve query performance by reducing the time required to retrieve data. The "cost" metric provides an estimate of the query execution complexity, allowing us to identify inefficient operations and optimize query performance. By monitoring and minimizing the cost value, we aim to improve the execution speed of queries, which is essential when working with large datasets like our temperature change data.

To test this hypothesis, we ran a simple script with several *'WHERE'* clauses to compare the query performance with and without additional indexes, attempting to confirm or refute our assumption.

```
SELECT *
FROM temperatures
WHERE countryname = 'Albania'
AND year BETWEEN 1970 and 2021;
```

Fig. 4. Select query used to test the improvement of adding additional indexes

After running the script, the results were gathered fast since the dataset is not that big, but the cost was 15. In the next experiments we tried to improve this value by lowering it.

1) Index over column countryname

Firstly, we decided to create an index over temperature column. The query is simple:

CREATE INDEX idx_countryname
ON temperatures(countryname);

The created index had a discernible impact on the cost value. It was lowered down to 2, making a very serious difference.

To test the effectiveness of all indexes we need to drop the previously added index.

2) Index over column year

The steps are the same, run the similar script as the one before, but change the column name to "year".

CREATE INDEX idx_countryname ON temperatures(year);

The applied index improved the cost value to 2 also.

3) Combined index of countryname and year

The created indexes can also be combined with multiple columns. We decided to test if the combination of the

countryname and year column in the newly created index will make any difference. The script we ran:

```
CREATE INDEX idx_countryname_year
  ON temperatures(countryname, year);
```

The results were surprising since the cost improved, but only to value of 3. This behaviour can be explained by the way the combined index is structured:

- The combined index is designed to optimize queries filtering by both *countryname* **and** *year* in the specific order of countryname first, then year,
- In cases where both columns are used in the query, the combined index **should** be more efficient. However, in this case, the database might still be using a suboptimal plan because of factors like the distribution of data, the order of columns in the index, or the way the optimizer estimates query costs.

Additionally, the query's use of the *BETWEEN* clause on year and the equality check on countryname might not align perfectly with how the combined index is structured, causing the query optimizer to resort to a less efficient plan, such as scanning the full index, rather than fully utilizing the combined index for both columns.

Thus, while the combined index is intended for use with both columns, individual indexes on each column (countryname and year) worked better in this specific case, resulting in a lower cost.

4) Hypothesis evaluation

After testing various indexing strategies, we can now evaluate our hypothesis that creating additional indexes on frequently queried columns improves query performance by reducing the query cost.

The initial query without indexes had a "cost" of 15, which indicated inefficient query execution. After creating individual indexes on countryname and year, the cost improved to 2, confirming that indexing each column separately had a positive effect on performance.

When we applied the combined index on *countryname* and *year*, we expected further improvement in performance. However, the cost only dropped to 3, which was less than the improvement achieved by the individual indexes. This suggests that while combined indexes are often efficient for queries involving both columns, they may not always outperform individual indexes, especially when the query optimizer does not fully optimize for both columns' combined structure.

Thus, our hypothesis was partially confirmed: indexing frequently queried columns does improve performance, but the combined index did not perform as expected, highlighting that in certain query patterns, individual indexes might be more effective than a combined index.

V. MATERIALIZED VIEWS

An important optimization strategy for improving query performance is to use a materialized view. This feature allows us to precalculate frequently requested values, which can save time when the data is required often, especially in the context of large datasets.

A materialized view stores the result of a query physically and periodically refreshes the data. This can significantly speed up query execution since it avoids recalculating the result every time a query is run.

We decided to create a materialized view for retrieving average temperature of each country. The test query before creating the materialized view resulted with time of 130ms in average.

```
1  SET TIMING ON;
2  SELECT countryname, ROUND(AVG(tempchange), 5) AS avg_temp_change
3  FROM temperatures
4  GROUP BY countryname;
5  SET TIMING OFF;
```

Fig. 5. Query to test the effect of materialized view

We ran this script to create the materialized view:

```
36 -- Priemerná zmena teploty za vsetky roky pre jednotlive krajiny
37 CREATE MATERIALIZED VIEW avg_temp_change_by_country
38  BUILD IMMEDIATE
39  REFRESH COMPLETE ON DEMAND
40  AS
41  SELECT countryname, ROUND(AVG(tempchange), 5) AS avg_temp_change
42  FROM temperatures
43  GROUP BY countryname;
```

Fig. 6. Materialized view for retrieving average temperature of each country

The breakdown of key words in the script:

- **BUILD IMMEDIATE** – this ensures that the materialized view is created and populated with data immediately when the command is executed, allowing it to be used right after its creation,
- **REFRESH COMPLETE ON DEMAND** – the materialized view can be manually refreshed as needed. The term complete refers to the fact that every row of the materialized view is completely recomputed during the refresh, ensuring that all the data is fully up to date with the latest changes from the source table (temperatures).
- **ROUND(AVG(tempchange), 5)** –

  This part of the query calculates the average temperature change for each country. The ROUND function is used to limit the result to 5 decimal places, ensuring that the computed average is precise and consistently formatted,
- **GROUP BY countryname** –

  The GROUP BY countryname clause organizes the data by each unique countryname. This means the query will calculate the average temperature change for each country individually, as opposed to calculating a single average for all countries combined.

```
SET TIMING ON;
SELECT countryname, avg_temp_change
FROM avg_temp_change_by_country;
SET TIMING OFF;
```

Fig. 7. Query to test the time cost of retrieving data from materialized view

Receiving data from the materialized view resulted with lower time, in average it took 110ms. This means that the effect of applying the materialized view saved 15% of time in comparison with the base query.

VI. PARTITIONING

Partitioning helps optimize large databases by splitting tables into smaller, more manageable sections. Each partition can be stored and accessed separately, improving query speed. This is especially useful for time-series data like global temperature records, where analysis often focuses on specific periods or regions. Below are the partitioning methods used in this study and their key benefits [10] [12] [13].

A. Range Partitioning

Range partitioning divides data based on specified value ranges. For the temperature dataset, the table was partitioned into four time intervals:

- 1970–1990 (p_1970_1990),
- 1991–2000 (p_1991_2000),
- 2001–2010 (p_2001_2010),
- 2011–2021 (p_2011_2021).

```
CREATE TABLE temperatures_range_partitioned (
    ID NUMBER,
    countryname VARCHAR2(100),
    year NUMBER,
    tempchange NUMBER
)
PARTITION BY RANGE (year) (
    PARTITION p_1970_1980 VALUES LESS THAN (1981),
    PARTITION p_1981_1990 VALUES LESS THAN (1991),
    PARTITION p_1991_2000 VALUES LESS THAN (2001),
    PARTITION p_2001_2010 VALUES LESS THAN (2011),
    PARTITION p_2011_2021 VALUES LESS THAN (2022)
);
```

Fig. 8. Range Partitioning by the year

The results can be checked by the same technique by running timer over the query that would be run on non-partitioned table and over this new partitioned table. The results didn't make a big difference since the dataset is small to make a significant difference.

B. Hash Partitioning

Hash partitioning distributes data across partitions using a hash function applied to a column. This method ensures even data distribution, reducing hotspots and improving load balancing.

Advantages:

- Ideal for queries filtering by non-sequential columns (e.g., country names).
- Enhances parallel read/write operations.
- Reduces contention in high-concurrency environments.

We used the hash partitioning shown in the figure below.

```
CREATE TABLE temperatures_hash_partitioned (
    ID NUMBER,
    countryname VARCHAR2(100),
    year NUMBER,
    tempchange NUMBER
)
PARTITION BY HASH (countryname)
PARTITIONS 4;
```

Fig. 9. Hash Partitioning

This partitioning is made by the countryname column.

C. Composite Partitioning

Composite partitioning is a combination of the basic data distribution methods; a table is partitioned by one data distribution method and then each partition is further subdivided into subpartitions using a second data distribution method. All subpartitions for a given partition represent a logical subset of the data.

```
CREATE TABLE temperatures_composite_partitioned(
    ID NUMBER,
    countryname VARCHAR2(100),
    year NUMBER,
    tempchange NUMBER
)
PARTITION BY RANGE (year)
SUBPARTITION BY HASH(countryname)
(
    PARTITION p_1970_1980 VALUES LESS THAN (1981),
    PARTITION p_1981_1990 VALUES LESS THAN (1991),
    PARTITION p_1991_2000 VALUES LESS THAN (2001),
    PARTITION p_2001_2010 VALUES LESS THAN (2011),
    PARTITION p_2011_2021 VALUES LESS THAN (2022)
);
```

Fig. 10. Composite Partitioning

Composite partitioning combines range and hash methods. The dataset is first partitioned by year ranges (RANGE), and each partition is further subdivided using hashing on countryname (HASH). We used this to optimize both time-based and region-based queries.[7]

Since our dataset is small, we were not able to measure the real impact of our partitionings, but there is a study that measured the improvement of 18-22% in horizontal database table partitioning. [4]

VII. DATA ANALYSIS AND FINDINGS

In this chapter, we will go through the interesting facts that were found by analysing the dataset.

A. Countries with the most rising temperatures (2000 - 2024)

For this purpose, we also created one more materialized view, which will store information about top 10 countries by highest average temperature rise judging by years 2000 to 2021.

```sql
CREATE MATERIALIZED VIEW top10_tempchange
BUILD IMMEDIATE
AS
SELECT COUNTRYNAME, avg_temp_change
FROM (
    SELECT COUNTRYNAME,
           AVG(TEMPCHANGE) AS avg_temp_change,
           ROW_NUMBER() OVER (ORDER BY AVG(TEMPCHANGE) DESC) AS rnk
    FROM temperatures
    WHERE YEAR BETWEEN 2000 AND 2021
      AND TEMPCHANGE IS NOT NULL
    GROUP BY COUNTRYNAME
)
WHERE rnk <= 10;
```

Fig. 11. Materialized view to store information about top 10 countries by temperature rise in years 2000-2021

Country	Avg. Temperature Change (°C)	Trend
Finland	1,61	↑ Warming
Cabo Verde	1,59	↑ Warming
Estonia, Rep. of	1,58	↑ Warming
Russian Federation	1,58	↑ Warming
Belarus, Rep. of	1,56	↑ Warming
Slovenia, Rep. of	1,56	↑ Warming
Austria	1,52	↑ Warming
Latvia	1,51	↑ Warming
Morocco	1,49	↑ Warming
Kazakhstan, Rep. of	1,48	↑ Warming

Fig. 12. Results of the top 10 average temperature change

The analysis reveals that Finland leads with the highest rise (+1.61°C), followed closely by Cabo Verde (+1.59°C) and Estonia (+1.58°C). Notably, Arctic and Eastern European nations dominate the list, with Russia, Belarus, and Latvia all showing increases above +1.50°C, underscoring accelerated warming in northern latitudes.

Key Observations:

1) Geographic Concentration: 7 of the top 10 countries are in Northern/Eastern Europe or adjacent regions, aligning with global observations of amplified Arctic warming.

2) Magnitude of Change: All listed countries exceeded an average rise of +1.48°C, far surpassing the global average (typically +0.8–1.2°C for this period).

3) Cabo Verde's Anomaly: As the sole tropical entry, its presence suggests localized climate shifts beyond polar amplification.

B. Detection of extreme temperature deviations

Climate variability often manifests in abrupt temperature shifts between consecutive years. To identify the most dramatic fluctuations, we analysed year-to-year changes across all countries using a SQL query that calculates the absolute difference in temperature between each year and its preceding year.

```sql
WITH TempData AS (
    SELECT CountryName, year, tempchange,
           LAG(tempchange) OVER (
               PARTITION BY CountryName
               ORDER BY year
           ) AS prev_year_temp
    FROM temperatures
)
SELECT CountryName, year, tempchange, prev_year_temp,
       tempchange - prev_year_temp AS change
FROM TempData
WHERE tempchange IS NOT NULL AND prev_year_temp IS NOT NULL
ORDER BY ABS(tempchange - prev_year_temp) DESC
FETCH FIRST 10 ROWS ONLY;
```

Fig. 13. SQL Query to get top 10 countries by the most significant interannual temperature change

The results are shown in the figure below:

	COUNTRYNAME	YEAR	TEMPCHANGE	PREV_YEAR_TEMP	CHANGE
1	Finland	1976	-0,872	1,898	-2,77
2	Kazakhstan, Rep. of	1996	-0,408	2,038	-2,446
3	Canada	2010	2,928	0,551	2,377
4	Greenland	1985	0,883	-1,462	2,345
5	Netherlands, The	1996	-0,783	1,537	-2,32
6	Moldova, Rep. of	1994	1,644	-0,666	2,31
7	Kazakhstan, Rep. of	1997	1,886	-0,408	2,294
8	Denmark	2014	2,681	0,401	2,28
9	Canada	1982	-0,665	1,566	-2,231
10	Finland	1988	0,579	-1,652	2,231

Fig. 14. Top 10 countries with the highest interannual temperature change

Key observations:

1) Finland (1976) the most severe drop

- -2.77°C year-to-year change (from +1.90°C in 1975 to -0.87°C in 1976),
- This shows that in Finland there was significantly colder year in comparison with the previous year.
- We tried to search for additional information on the internet if there was a reason for this temperature change, but there are none, this means that the change might be caused by the combination of global warming and also the fact that the previous year was warmer.

2) Geographic Patterns:

- Northern/Arctic regions (Finland, Canada, Greenland, Denmark) dominate the list, suggesting heightened volatility at higher latitudes.

C. Countries with the all time highest average temperature rise(1970 - 2021)

In the graph below we can see countries that had the biggest average temperature rise for all the years. If there are so many countries that have temperature higher, we shouldn't doubt global warming.

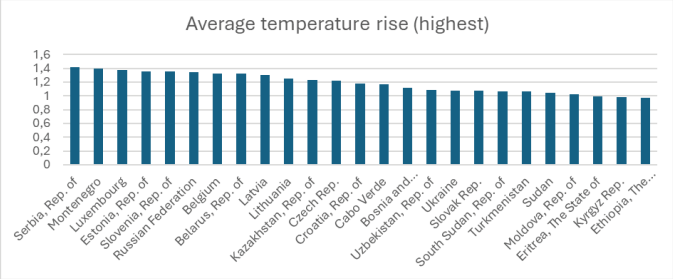

Fig. 15. Countries with the all-time highest average temperature rise (1970 - 2021)

To proof that this rise is not just random, we can take 25 countries with lowest temperature rise/highest down of the temperature. We can see that there is only 1 country which temperature went down by the years, all other countries have higher temperature.

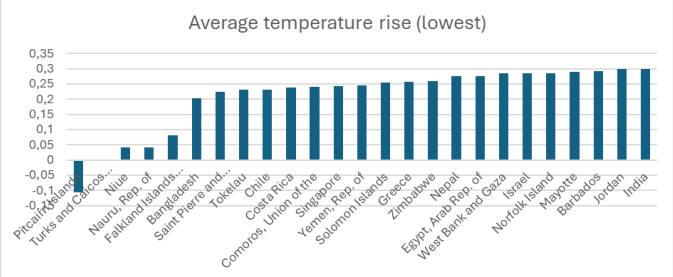

Fig. 16. Countries with all-time lowest average temperatures rise (1970 - 2021)

As we can see, hardly any countries have experienced a decrease in temperature. Nearly every nation in the world is facing the phenomenon known as global warming.

D. *Observing highest deviations in temperatures*

This subchapter focuses on looking for countries that have the lowest stability of the temperature. Standard deviation is a good statistical tool which can show us what we are looking for.

The aggregational function STDDEV in oracle achieves the required result.

```
SELECT COUNTRYNAME AS "Country",
    ROUND(STDDEV(TEMPCHANGE), 2) AS "SD"
FROM temperatures
WHERE TEMPCHANGE IS NOT NULL
GROUP BY COUNTRYNAME
ORDER BY "SD" DESC
FETCH FIRST 25 ROWS ONLY;
```

Fig. 17. Getting top 25 countries with highest standard deviation of temperature

The results are shown in graph below:

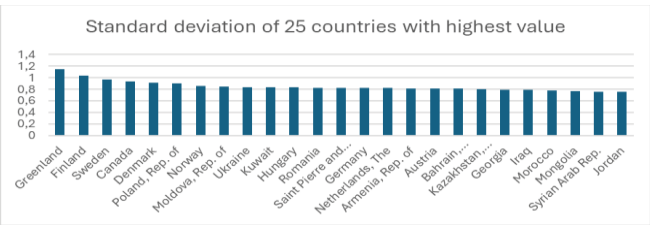

Fig. 18 Result graph showing top 25 countries with the highest standard deviation of temperature

E. *Global perspective on the issue of rising temperatures*

Until now we focused on local changes, but in the international context the results are way more accurate. In the dataset are also already calculated data for the whole world values for each year. In the graph below we can see how the temperature was rising until 2022.

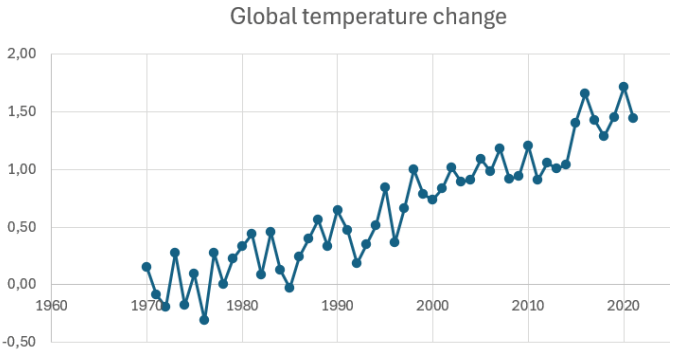

Fig. 19. Chart - global temperature rise

As we can see the temperature is overall rising. This indicates that the global warming is true. In the chart we can see that until years 1990 the temperature wasn't rising that much as in the later years. This is caused by the fact that the baseline temperature was established from the average temperature from years 1951 to 1980, so the temperature around those years couldn't rise as much as in the years later.

VIII. Conclusion

This study demonstrates how advanced database optimization techniques can significantly enhance the efficiency of data processing and analysis in climate change research. By implementing indexing, materialized views, and partitioning strategies in Oracle Database, we achieved measurable improvements in query performance when analysing large-scale global temperature datasets. Although the initial dataset was relatively small, our experiments confirmed that these optimization methods reduce query execution times, enabling faster and more scalable data analysis.

Future research directions could include applying these techniques to even larger and more complex climate datasets, integrating machine learning models for predictive climate analysis, and incorporating real-time data streams to support dynamic, up-to-date environmental monitoring. Such advancements could further empower researchers to extract actionable insights with greater speed and accuracy.

Our findings confirmed the ongoing trend of global warming, with Arctic and Eastern European countries experiencing the most significant temperature increases. The study also identified extreme interannual variations and highlighted the importance of database optimizations in managing large datasets effectively.

Acknowledgment

This paper was also supported by the **VEGA 1/0192/24** project - *Developing and applying advanced techniques for*

efficient processing of large-scale data in the intelligent transport systems environment.

REFERENCES

[1] M. A. K, Lovru, "*All Countries Temperature Statistics 1970 – 2021*": https://www.kaggle.com/datasets/mdazizulkabirlovlu/all-countries-temperature-statistics-1970-2021

[2] International Monetary Fund, Climate Change Data "*Annual surface temperature Change*": https://climatedata.imf.org/pages/climatechange-data

[3] Geeksforgeeks.org *"Indexing in Databases – Set 1*": https://www.geeksforgeeks.org/indexing-in-databases-set-1/

[4] R. Elmasri and S. B. Navathe, "*Fundamentals of Database Systems (7th ed.)*". Pearson, 2016

[5] Sciencedirect.com,."*Range Partitioning*": https://www.sciencedirect.com/topics/computer-science/range-partitioning

[6] Dremio.com,."*Hash Partitioning*": https://www.dremio.com/wiki/hash-partitioning/

[7] R. Greenwald, R. Stackowiak, and J. Stern, "*Oracle Essentials: Oracle Database 12c"*, O'Reilly Media, 2013.

[8] D. Kuhn, and T. Kyte, "*Expert Oracle Database Architecture: Techniques and Solutions for High Performance and Productivity."* Apress, 2021.

[9] M. Kvet, "*Developing Robust Date and Time Oriented Applications in Oracle Cloud: A comprehensive guide to efficient Date and time management in Oracle Cloud"*, Packt Publishing, 2023, ISBN: 978-1804611869

[10] A. Nuijten, A. Barel, *"Modern Oracle Database Programming: Level Up Your Skill Set to Oracle's Latest and Most Powerful Features in SQL, PL/SQL, and JSON"*, Apress, 2023

[11] M. Malcher and D. Kuhn, "*Pro Oracle Database 23c Administration: Manage and Safeguard Your Organization's Data*." Berkeley, CA: Apress, 2024, ISBN: 978-1-4842-9898-5.

[12] Erasmus+ project EverGreen dealing with the complex data analytics: https://evergreen.uniza.sk/

[13] Docs.oracle.com,."Composite Partitioning": https://docs.oracle.com/database/121/VLDBG/GUID-BE424ACC-F746-4CA8-973C-F578CF98FF10.htm