Assignment 03 – CSE 412
Bhavya Patel – ASU ID: 1225740997

Part A

A1. Write the SQL statements to create `SuppliersView` and `SupplierInfoView`. (SQL Statement)

```
CREATE VIEW SuppliersView AS
SELECT S_NAME
FROM SUPPLIER;
```

```
CREATE VIEW SupplierInfoView AS
SELECT S.S_NAME, COUNT(PS.PS_PARTKEY) AS num_parts
FROM SUPPLIER S
JOIN PARTSUPP PS ON S.S_SUPPKEY = PS.PS_SUPPKEY
GROUP BY S.S_NAME;
```

A2. Retrieve the total number of parts supplied by '`Supplier#000000001`' using `SupplierInfoView`. (SQL Statement)

```
SELECT num_parts
FROM SupplierInfoView
WHERE S_NAME = 'Supplier#000000001';
```

A3. What permissions should be given to your secretary to allow them to remove suppliers from the Suppliers table? Explain and suggest the SQL Statement.

```
GRANT DELETE ON SUPPLIER TO Secretary;
```

To remove suppliers, the secretary should be granted permission to delete rows from the supplier table, which is done using GRANT DELETE on that table.

A4. Write the SQL command that allows your secretary to grant others read access to `SuppliersView`. (SQL Statement)

```
GRANT SELECT ON SuppliersView TO Secretary WITH GRANT OPTION;
```

A5. If your secretary grants Todd read access to `SupplierInfoView` and later quits, what command should you use to revoke their privileges? What happens to Todd's access? Explain and provide the SQL Statement

```
REVOKE SELECT ON SupplierInfoView FROM Secretary CASCADE;
```

Using the CASCADE option when revoking the secretary's access, it would also remove Todd's or anyone else's access that were granted to others by the secretary since they depended on the secretary's grant option.

A6. Since you are going on an extended vacation, authorize your boss, Joe, to read and modify the Suppliers table, `SuppliersView`, and `SupplierInfoView`, ensuring he can also delegate authority. Provide the appropriate SQL statements.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON SUPPLIER TO Joe WITH GRANT OPTION;

GRANT SELECT, INSERT, UPDATE, DELETE ON SuppliersView TO Joe WITH GRANT OPTION;

GRANT SELECT, INSERT, UPDATE, DELETE ON SupplierInfoView TO Joe WITH GRANT OPTION;
```

Part B

B1. Given the functional dependencies:

1. *CustID → Name, Address, State, Zip*
2. *BookID → Title, Author, Price, Category*
3. *OrderID → CustID, BookID, ShipDate*

Determine the Highest Normal Form for the **Order Table**? Justify your answer by analyzing partial and transitive dependencies.

The Order table meets the requirements for 2NF but not for 3NF. OrderID uniquely identifies every other field in the table, it serves as the primary key. Because the key consists of just one attribute, there are no partial dependencies, meaning the table satisfies both 1NF and 2NF. However, it violates 3NF because it contains transitive dependencies:

- OrderID –> CustID –> Name, Address, State, Zip
- OrderID –> BookID –> Title, Author, Price, Category

Here, several non-key attributes depend on CustID or BookID rather than depending directly on OrderID, table violates rule of 3NF.
The highest normal form of this table is 2NF.

B2. Given the functional dependencies:

1. *BookID, WarehouseID → Quantity, ShelfLocation*
2. *WarehouseID → State*

Determine the Highest Normal Form for the ***Inventory Table***. Justify your answer based on partial and transitive dependencies

The Order table meets the requirements for 1NF but not for 2NF. The Primary Key for this table is the composite pair (BookID, WarehouseID). A table is in 2NF only if there are no partial dependencies. But WarehouseID → State shows that State depends only on WarehouseID, which is just one part of the composite primary key. Because a partial dependency exists, the table fails 2NF and remains in 1NF.
The highest normal form of this table is 2NF.

B3. Given the functional dependencies:

1. *CustID → Name, Address, State, Zip*
2. *BookID → Title, Author, Price, Category*
3. *OrderID → CustID, BookID, ShipDate*

Decompose the Order table into multiple relations conforming to BCNF.
List all decomposed relations.

Customer (CustID, Name, Address, State, Zip)
Book (BookID, Title, Author, Price, Category)
Order (OrderID, CustID, BookID, ShipDate)

B4. Given the functional dependencies:

1. *BookID, WarehouseID → Quantity, ShelfLocation*
2. *WarehouseID → State*

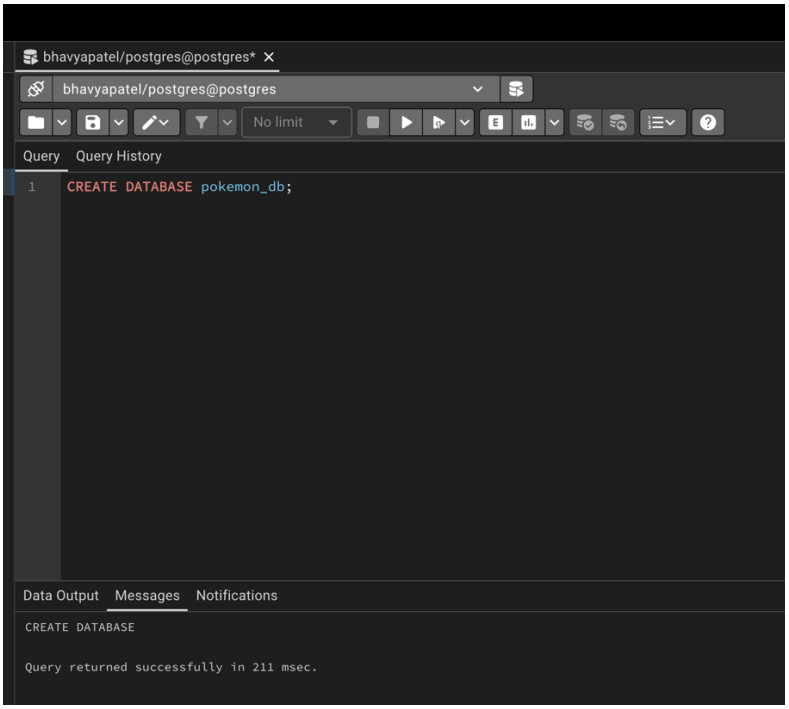
Decompose the ***Inventory table*** into multiple relations conforming to BCNF. List all decomposed relations.

Warehouse (WarehouseID, State)
Inventory (BookID, WarehouseID, Quantity, ShelfLocation)

Part C

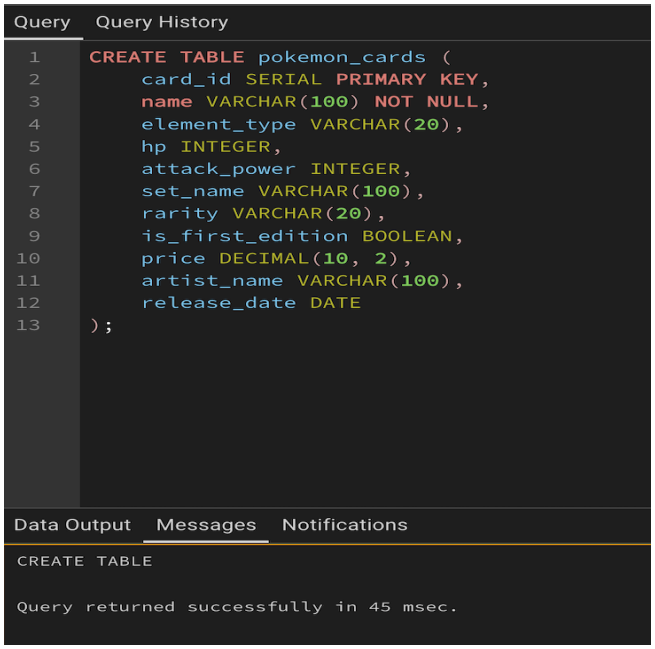
C1. Database and Table Creation

- Create a **new PostgreSQL database** with the name of your choice. What SQL command would you use to create this database? Provide a snapshot of the SQL statement and the database created.




```
CREATE DATABASE pokemon_db;
```

- Create a **table** with at least **10 columns**, including a **primary key** and a few categorical and numerical attributes. Provide a snapshot of the SQL statement and explain it.




```
CREATE TABLE pokemon_cards (
    card_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    element_type VARCHAR(20),
    hp INTEGER,
    attack_power INTEGER,
    set_name VARCHAR(100),
    rarity VARCHAR(20),
    is_first_edition BOOLEAN,
    price DECIMAL(10, 2),
    artist_name VARCHAR(100),
    release_date DATE
);
```

- Populate the table with at least **2 million rows** using random or meaningful data. Provide a snapshot of the SQL statement and explain it.

Query | Query History

```
INSERT INTO pokemon_cards (
    name, element_type, hp, attack_power, set_name, rarity,
    is_first_edition, price, artist_name, release_date
)
SELECT
    'Pokemon ' || i,
    (ARRAY['Fire', 'Water', 'Grass', 'Electric', 'Psychic', 'Fighting', 'Dark', 'Steel'])[floor(random() *
    floor(random() * 300 + 30)::int,
    floor(random() * 200 + 10)::int,
    (ARRAY['Base Set', 'Jungle', 'Fossil', 'Team Rocket', 'Neo Genesis', 'Skyridge'])[floor(random() * 6 +
    (ARRAY['Common', 'Uncommon', 'Rare', 'Holo Rare', 'Secret Rare'])[floor(random() * 5 + 1)],
    (random() > 0.9),
    (random() * 500 + 1)::decimal(10, 2),
    'Artist ' || floor(random() * 50 + 1),
    CURRENT_DATE - (floor(random() * 9000) || ' days')::interval
FROM generate_series(1, 2000000) AS s(i);
```

Data Output | Messages | Notifications

```
INSERT 0 2000000

Query returned successfully in 9 secs 986 msec.
```

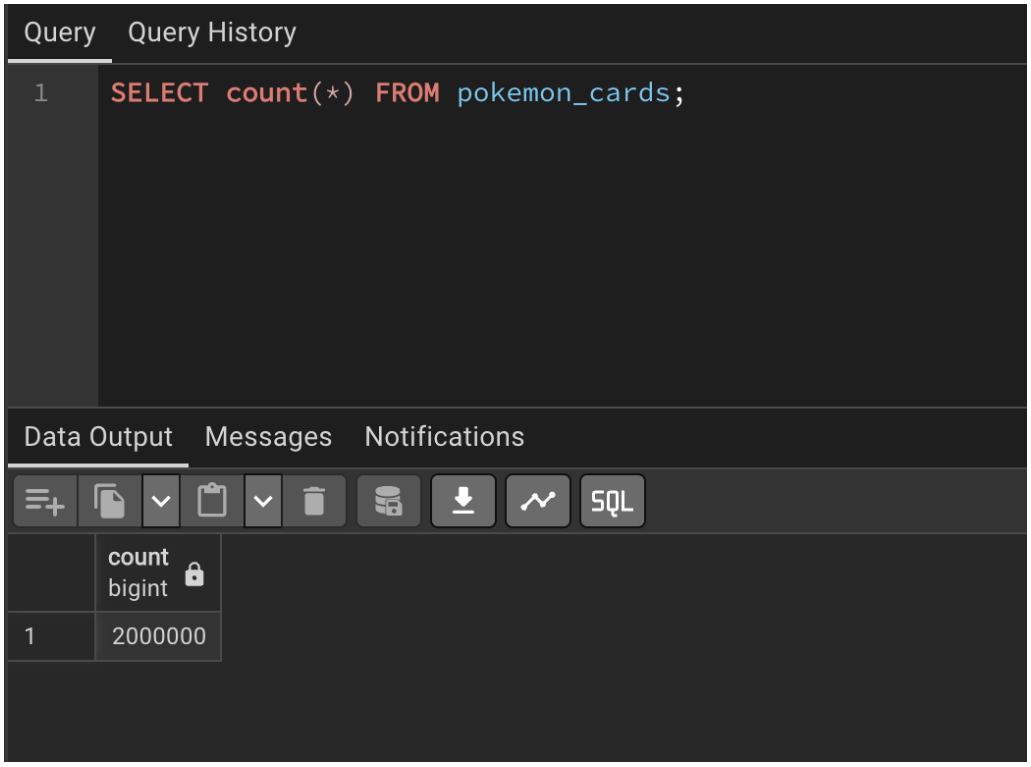
generate_series(1, 2000000): This creates a temporary set of 2 million rows, acting as the loop driver.

random() & Arrays: By defining arrays (e.g., for element_type) and picking a random index, we ensure the column has low cardinality (repeated values), which is essential for testing index effectiveness later.

concatenation (||): Used to create unique names (Pokemon 1... Pokemon 2000000) so not every column is a duplicate.

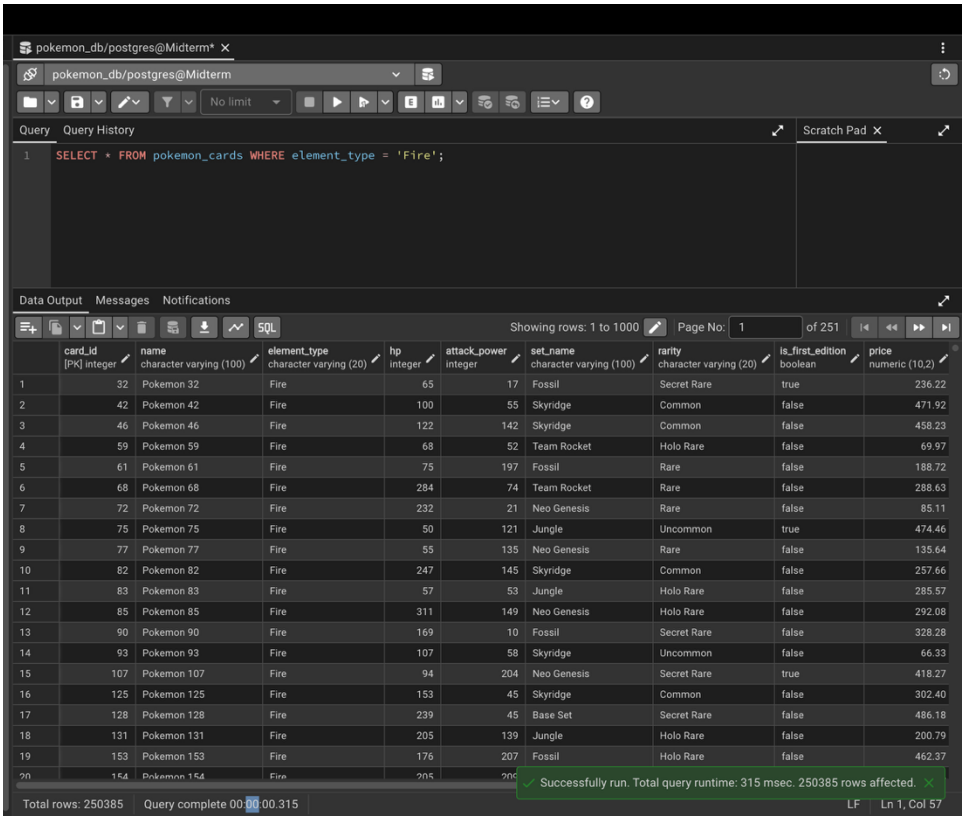
• Write a query to verify that the table contains **2 million rows**. Provide a snapshot of the SQL statement used and the output.

SELECT count(*) FROM pokemon_cards;

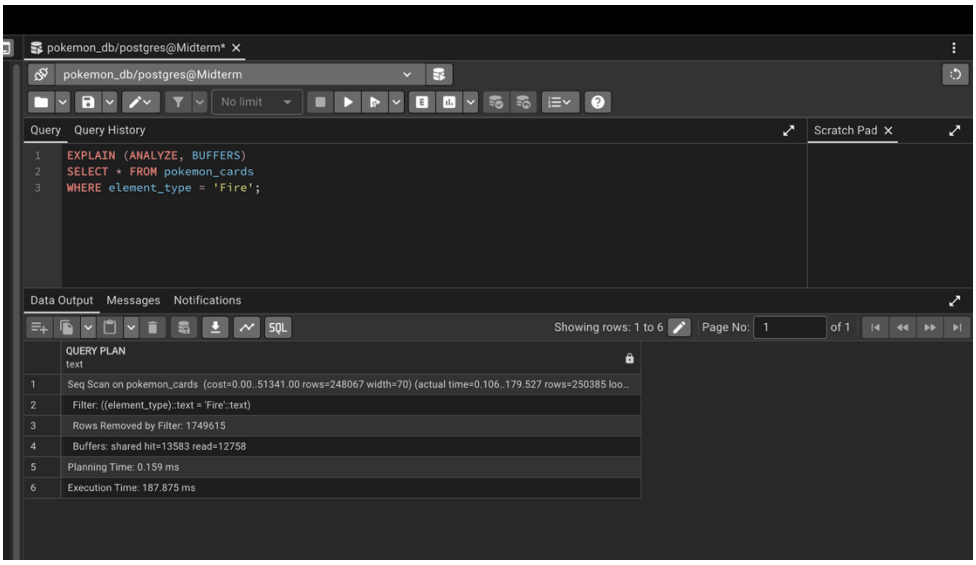


C2. Query Cost Analysis

- Choose a **query** that includes a WHERE condition filtering on a specific column.
- Use appropriate SQL commands to analyze the query execution plan (QEP) and answer the following questions.
- Provide a snapshot of the query used to filter data.
- Show a snapshot of the query result after applying the filter.
- Capture the query execution plan (QEP) output.
- Identify the type of scan used. Briefly explain its significance.
- Compare the estimated and actual costs from the execution plan and explain their meaning.
- Indicate the number of rows removed by the filter and briefly explain its impact.
- Explain what Buffers: shared hit/read represents in the execution plan.
- Calculate the query **response time** using: Response time=CPU time + I/O time



	card_id [PK] integer	name character varying (100)	element_type character varying (20)	hp integer	attack_power integer	set_name character varying (100)	rarity character varying (20)	is_first_edition boolean	price numeric (10,2)
1	32	Pokemon 32	Fire	65	17	Fossil	Secret Rare	true	236.22
2	42	Pokemon 42	Fire	100	55	Skyridge	Common	false	471.92
3	46	Pokemon 46	Fire	122	142	Skyridge	Common	false	458.23
4	59	Pokemon 59	Fire	68	52	Team Rocket	Holo Rare	false	69.97
5	61	Pokemon 61	Fire	75	197	Fossil	Rare	false	188.72
6	68	Pokemon 68	Fire	284	74	Team Rocket	Rare	false	288.63
7	72	Pokemon 72	Fire	232	21	Neo Genesis	Rare	false	85.11
8	75	Pokemon 75	Fire	50	121	Jungle	Uncommon	true	474.46
9	77	Pokemon 77	Fire	55	135	Neo Genesis	Rare	false	135.64
10	82	Pokemon 82	Fire	247	145	Skyridge	Common	false	257.66
11	83	Pokemon 83	Fire	57	53	Jungle	Holo Rare	false	285.57
12	85	Pokemon 85	Fire	311	149	Neo Genesis	Holo Rare	false	292.08
13	90	Pokemon 90	Fire	169	10	Fossil	Secret Rare	false	328.28
14	93	Pokemon 93	Fire	107	58	Skyridge	Uncommon	false	66.33
15	107	Pokemon 107	Fire	94	204	Neo Genesis	Secret Rare	true	418.27
16	125	Pokemon 125	Fire	153	45	Skyridge	Common	false	302.40
17	128	Pokemon 128	Fire	239	45	Base Set	Secret Rare	false	486.18
18	131	Pokemon 131	Fire	205	139	Jungle	Holo Rare	false	200.79
19	153	Pokemon 153	Fire	176	207	Fossil	Holo Rare	false	462.37
20	154	Pokemon 154	Fire	205	200				



It used a Sequential Scan (Seq Scan) meaning PostgreSQL scanned every row in the table from top to bottom. A Seq Scan is slow on large tables because it must read all rows, even though the filter only keeps a small portion of them.

Estimated vs actual

- Estimated Cost (0.00..51341.00)
    - Calculated by the PostgreSQL planner before running the query.
    - Represents predicted CPU + I/O effort.
    - 0.00 = startup cost
    - 51341.00 = total estimated work to complete the scan.
- Actual Time (0.106..179.527 ms)
    - Measured during execution.
    - 0.106 ms = first row returned
    - 179.527 ms = total time to finish scanning all rows
- The planner's estimates and the actual timings are close, meaning PostgreSQL's statistics about the table are accurate and the planner predicted the work well.

Rows removed

- Rows Removed by Filter: 1749615
- Rows Returned: 250385
- The database scanned 2,000,000 rows, removed 1,749,615 rows, and kept only cards with the element 'Fire' (~250k rows).
- A high number of removed rows indicates: The filter is highly selective, Sequential scans do unnecessary work.

Buffers: shared hit=13583 read=12758

- Buffers: shared hit/read shows how PostgreSQL accessed data pages during the query.
- shared hit = 13,583 means PostgreSQL found these pages already stored in memory. These are fast, RAM-level accesses and do not require disk I/O.
- shared read = 12,758 means PostgreSQL had to fetch these pages from disk because they were not already in memory. Disk reads are much slower and contribute significantly to query time.

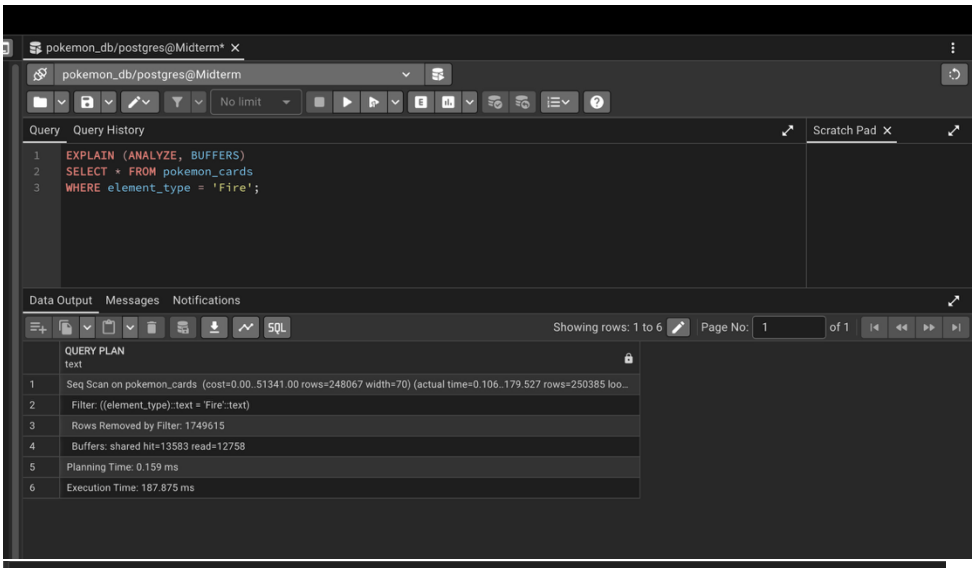
The query response time

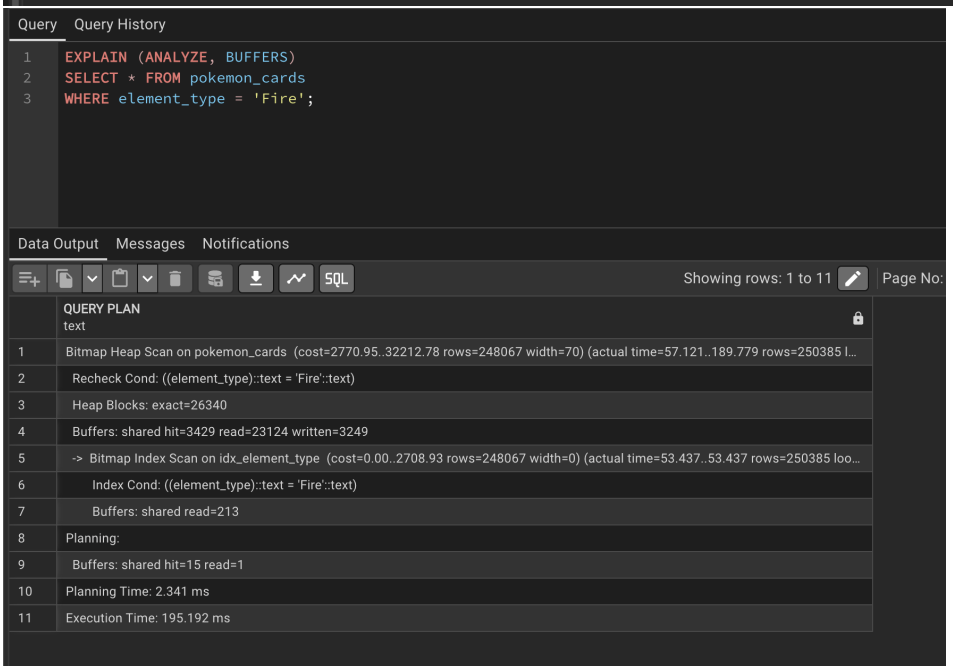
- According to the formula: Response time = CPU time + I/O time and the data from ANALYZE showing execution time.
- The total response time for the query is: 187.875 ms.

C3. Query Optimization

Perform a comparative analysis of query execution before and after optimization, using techniques like indexing. Execute a WHERE clause query without indexing and then rerun the same query with indexing applied.
Answer the following questions:

- Provide snapshots of the query results and execution plans for both cases. ○ Describe how the execution plan differs before and after optimization. ○ Did the query cost decrease? Explain briefly.
- Has the execution time improved? Explain briefly ○ Is PostgreSQL now using an Index Scan instead of a Sequential Scan? Explain the difference.
- How did the values for Buffers (shared hit, read, etc.) change? Explain briefly.





- Before and after
    - Before optimization, it performed a Sequential Scan, meaning it scanned all 2,000,000 rows in the table and filtered each row one by one.
    - After optimization, it switched to a Bitmap Index Scan + Bitmap Heap Scan, which is an index-driven strategy.
    - Instead of reading the whole table, the database used the index on element_type to quickly locate all “Fire” rows and then fetched only the matching heap blocks.
- Cost
    - Yes, the cost decreased significantly.
        - Before: cost= 0.00..51341.00
        - After: cost=2770.95..32212.78
    - The upper cost value dropped from 51,341 to 32,212, which means the planner expected the optimized plan to require less computing resources.
    - This decrease occurs because the index allows PostgreSQL to avoid continuously scanning the entire 2M-row table.
- Time
    - No, it did not improve, in fact it slightly increased:
        - Before: 187.875 ms
        - After: 195.192 ms
    - Indexes don’t always help when the filter selects a large percentage of rows (in our case, ~250,000 "Fire" cards).
    - Bitmap scans introduce overhead, and when a significant chunk of the table needs to be returned anyway, the raw speed of a sequential scan can sometimes be faster than jumping around the index.
- Index scan
    - PostgreSQL is now using a Bitmap Index Scan combined with a Bitmap Heap Scan, instead of a Sequential Scan.
    - Sequential Scan (Before)
        - Reads every row in the table.
        - Applies the filter to all 2 million rows.
        - Efficient only when most of the table is needed.
    - Bitmap Index Scan + Bitmap Heap Scan (After)
        - Uses the index idx_element_type to quickly find matching rows (element_type = 'Fire').
        - Fetches only the required heap blocks.
        - Efficient for filters that select a specific, smaller subset of rows.
- Buffers
    - Before: Buffers: shared hit=13583 read=12758
    - PostgreSQL had a balanced mix of reading from memory (hits) and disk (reads) during the sequential scan.
    - After: Buffers: shared hit=3429 read=23124
    - Far fewer shared hits (because we’re no longer scanning every page in order and benefiting from the OS cache as much).

- Much higher shared reads (23,124 vs 12,758). PostgreSQL had to fetch many pages from the disk to satisfy the Bitmap Heap Scan.
- The new plan caused more disk I/O ("reads"), which explains why the execution time didn't improve.

C4. Parallel Query Execution

Implement parallel execution and rerun the query used in C3 (Query Optimization) to analyze its impact.
Answer the following questions:

- How many workers were planned vs. how many were launched? o How does parallel execution affect query cost and response time?
- Compare the execution time before and after enabling parallel execution.
- Capture and submit snapshots of the query execution plan (QEP) with and without parallel execution.