SOMAIYA VIDYAVIHAR UNIVERSITY

RDBMS IA_2 23-24

Comparative Analysis of Join Algorithms:- Tuple, Block, Sort, & Index-based

B/2

Department of Computer Engineering

Nikhil Patil
16010122136
Hyder Presswala
16010122151
Vedant Rathi
16010122154
Ronak Rathod
16010122156

Date of submission:-15/04/2024.

Title of the implementation:- Join Algorithm Performance

Introduction:-

- We will be using Python for implementing all algorithms .
- We will test the code with two different sizes datasets :-
    - In first data set we generate dataset of Size from 1 to 5000. [It goes from 1-1000, then 1-2000, then 1-3000, then 1-4000 and last 1-5000.]
    - In Second dataset we generate the dataset of Size from 1 to 10000[It goes from 1-1000, then 1-2000, then 1-3000, then 1-4000, then 1-5000, 1-6000, then 1-7000, then 1-8000, then 1-9000 and last 1-10000.]
- We have Separate code for Comparison and Analysis :-
    - For Comparation we have a multi-line graph in which plotting of all methods are done on same Graph.
    - And for Analysis Single Line graph of each method in which plotting of all methods are done on different Graph.

Objective:-

We compare the performance of different implementations with different sizes of data sets and show results using comparative graphs for each dataset.

Implementation Details:-

Code for dataset from 1-5000:-

```
import time

import random
import matplotlib.pyplot as plt

# Tuple-based Join Algorithm
def tuple_join(A, B):
    result = []
    for a in A:
        for b in B:
            if a[0] == b[0]:
                result.append(a + b)
    return result

# Block-based Join Algorithm
def block_join(A, B, block_size):
    result = []
    for i in range(0, len(A), block_size):
        for j in range(0, len(B), block_size):
            for a in A[i:i+block_size]:
                for b in B[j:j+block_size]:
                    if a[0] == b[0]:
                        result.append(a + b)
    return result

# Sort-based Join Algorithm
def sort_join(A, B):
    A.sort(key=lambda x: x[0])
    B.sort(key=lambda x: x[0])
    result = []
    i = j = 0
    while i < len(A) and j < len(B):
        if A[i][0] < B[j][0]:
            i += 1
        elif A[i][0] > B[j][0]:
            j += 1
        else:
            result.append(A[i] + B[j])
            j += 1
    return result

# Index-based Join Algorithm
def index_join(A, B):
```

```
    index = {}
    for b in B:
        index[b[0]] = b
    result = []
    for a in A:
        if a[0] in index:
            result.append(a + index[a[0]])
    return result

# Function to generate synthetic datasets
def generate_dataset(size):
    return [(i, random.randint(1, 100)) for i in range(size)]

# Function to perform time analysis
def time_analysis(algorithm, dataset_sizes):
    times = []
    for size in dataset_sizes:
        A = generate_dataset(size)
        B = generate_dataset(size)
        start_time = time.time()
        algorithm(A, B)
        end_time = time.time()
        times.append(end_time - start_time)
    return times

# Dataset sizes to be tested
dataset_sizes = [1000, 2000, 3000, 4000, 5000]

# Perform time analysis for each algorithm
tuple_times = time_analysis(tuple_join, dataset_sizes)
block_times = time_analysis(lambda A, B: block_join(A, B, 100),
dataset_sizes)  # Using block size of 100
sort_times = time_analysis(sort_join, dataset_sizes)
index_times = time_analysis(index_join, dataset_sizes)

# Plotting results
plt.plot(dataset_sizes, tuple_times, label='Tuple-based Join')
plt.plot(dataset_sizes, block_times, label='Block-based Join')
plt.plot(dataset_sizes, sort_times, label='Sort-based Join')
plt.plot(dataset_sizes, index_times, label='Index-based Join')
plt.xlabel('Dataset Size')
plt.ylabel('Time (seconds)')
plt.title('Comparative Analysis of Join Algorithms')
plt.legend()
plt.show()
```

Code for dataset from 1-10000:-

```
import time
import random
import matplotlib.pyplot as plt

# Tuple-based Join Algorithm
def tuple_join(A, B):
    result = []
    for a in A:
        for b in B:
            if a[0] == b[0]:
                result.append(a + b)
    return result

# Block-based Join Algorithm
def block_join(A, B, block_size):
    result = []
    for i in range(0, len(A), block_size):
        for j in range(0, len(B), block_size):
            for a in A[i:i+block_size]:
                for b in B[j:j+block_size]:
                    if a[0] == b[0]:
                        result.append(a + b)
    return result

# Sort-based Join Algorithm
def sort_join(A, B):
    A.sort(key=lambda x: x[0])
    B.sort(key=lambda x: x[0])
    result = []
    i = j = 0
    while i < len(A) and j < len(B):
        if A[i][0] < B[j][0]:
            i += 1
        elif A[i][0] > B[j][0]:
            j += 1
        else:
            result.append(A[i] + B[j])
            j += 1
    return result

# Index-based Join Algorithm
def index_join(A, B):
    index = {}
    for b in B:
        index[b[0]] = b
    result = []
```

```
    for a in A:
        if a[0] in index:
            result.append(a + index[a[0]])
    return result

# Function to generate synthetic datasets
def generate_dataset(size):
    return [(i, random.randint(1, 100)) for i in range(size)]

# Function to perform time analysis
def time_analysis(algorithm, dataset_sizes):
    times = []
    for size in dataset_sizes:
        A = generate_dataset(size)
        B = generate_dataset(size)
        start_time = time.time()
        algorithm(A, B)
        end_time = time.time()
        times.append(end_time - start_time)
    return times

# Dataset sizes to be tested
dataset_sizes = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

# Perform time analysis for each algorithm
tuple_times = time_analysis(tuple_join, dataset_sizes)
block_times = time_analysis(lambda A, B: block_join(A, B, 100),
dataset_sizes)  # Using block size of 100
sort_times = time_analysis(sort_join, dataset_sizes)
index_times = time_analysis(index_join, dataset_sizes)

# Plotting results
plt.plot(dataset_sizes, tuple_times, label='Tuple-based Join')
plt.plot(dataset_sizes, block_times, label='Block-based Join')
plt.plot(dataset_sizes, sort_times, label='Sort-based Join')
plt.plot(dataset_sizes, index_times, label='Index-based Join')
plt.xlabel('Dataset Size')
plt.ylabel('Time (seconds)')
plt.title('Comparative Analysis of Join Algorithms')
plt.legend()
plt.show()
```

Code for Analytic:-

Code for dataset from 1-5000:-

```
import time
import random
import matplotlib.pyplot as plt

# Tuple-based Join Algorithm
def tuple_join(A, B):
    result = []
    for a in A:
        for b in B:
            if a[0] == b[0]:
                result.append(a + b)
    return result

# Block-based Join Algorithm
def block_join(A, B, block_size):
    result = []
    for i in range(0, len(A), block_size):
        for j in range(0, len(B), block_size):
            for a in A[i:i+block_size]:
                for b in B[j:j+block_size]:
                    if a[0] == b[0]:
                        result.append(a + b)
    return result

# Sort-based Join Algorithm
def sort_join(A, B):
    A.sort(key=lambda x: x[0])
    B.sort(key=lambda x: x[0])
    result = []
    i = j = 0
    while i < len(A) and j < len(B):
        if A[i][0] < B[j][0]:
            i += 1
        elif A[i][0] > B[j][0]:
            j += 1
        else:
            result.append(A[i] + B[j])
            j += 1
    return result

# Index-based Join Algorithm
def index_join(A, B):
```

```
    index = {}
    for b in B:
        index[b[0]] = b
    result = []
    for a in A:
        if a[0] in index:
            result.append(a + index[a[0]])
    return result

# Function to generate synthetic datasets
def generate_dataset(size):
    return [(i, random.randint(1, 100)) for i in range(size)]

# Function to perform time analysis
def time_analysis(algorithm, dataset_sizes):
    times = []
    for size in dataset_sizes:
        A = generate_dataset(size)
        B = generate_dataset(size)
        start_time = time.time()
        algorithm(A, B)
        end_time = time.time()
        times.append(end_time - start_time)
    return times

# Dataset sizes to be tested
dataset_sizes = [1000, 2000, 3000, 4000, 5000]

# Perform time analysis for each algorithm
tuple_times = time_analysis(tuple_join, dataset_sizes)
block_times = time_analysis(lambda A, B: block_join(A, B, 100),
dataset_sizes)  # Using block size of 100
sort_times = time_analysis(sort_join, dataset_sizes)
index_times = time_analysis(index_join, dataset_sizes)

# Plotting results for all join algorithms in the same window
fig, axs = plt.subplots(2, 2, figsize=(12, 8))

# Tuple-based Join
axs[0, 0].plot(dataset_sizes, tuple_times, label='Tuple-based Join')
axs[0, 0].set_title('Tuple-based Join')
axs[0, 0].set_xlabel('Dataset Size')
axs[0, 0].set_ylabel('Time (seconds)')

# Block-based Join
axs[0, 1].plot(dataset_sizes, block_times, label='Block-based Join')
axs[0, 1].set_title('Block-based Join')
```

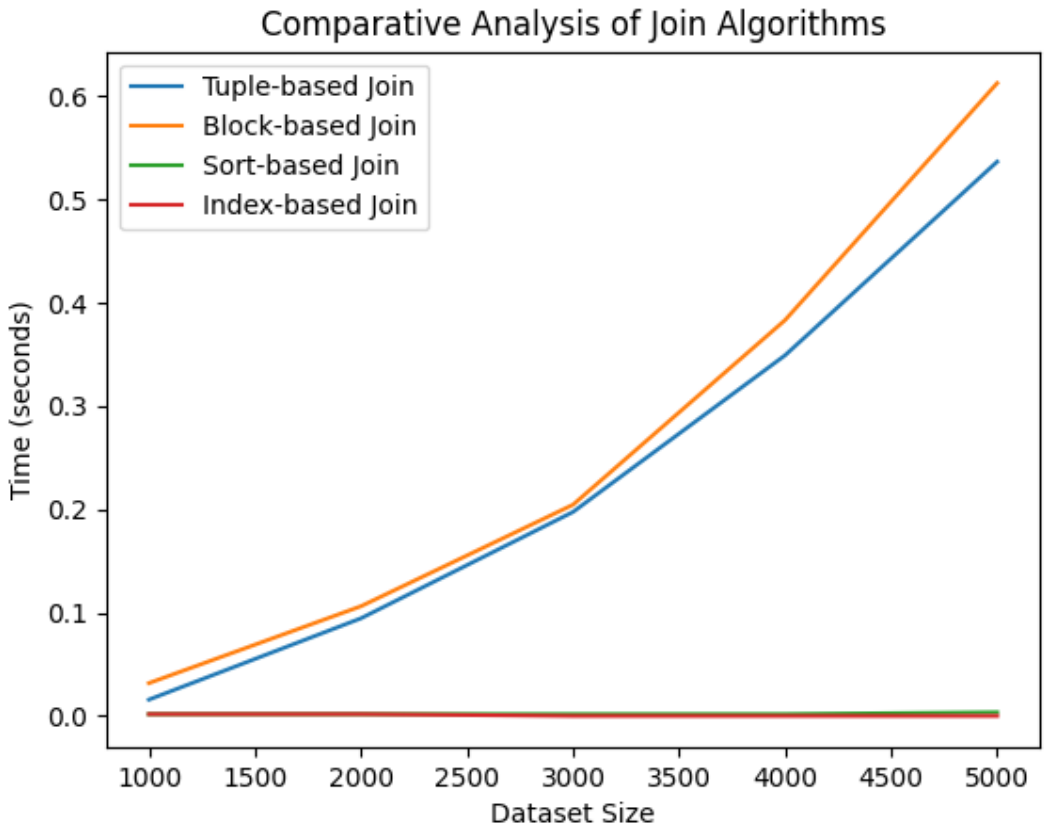
Code for dataset from 1-10000:-

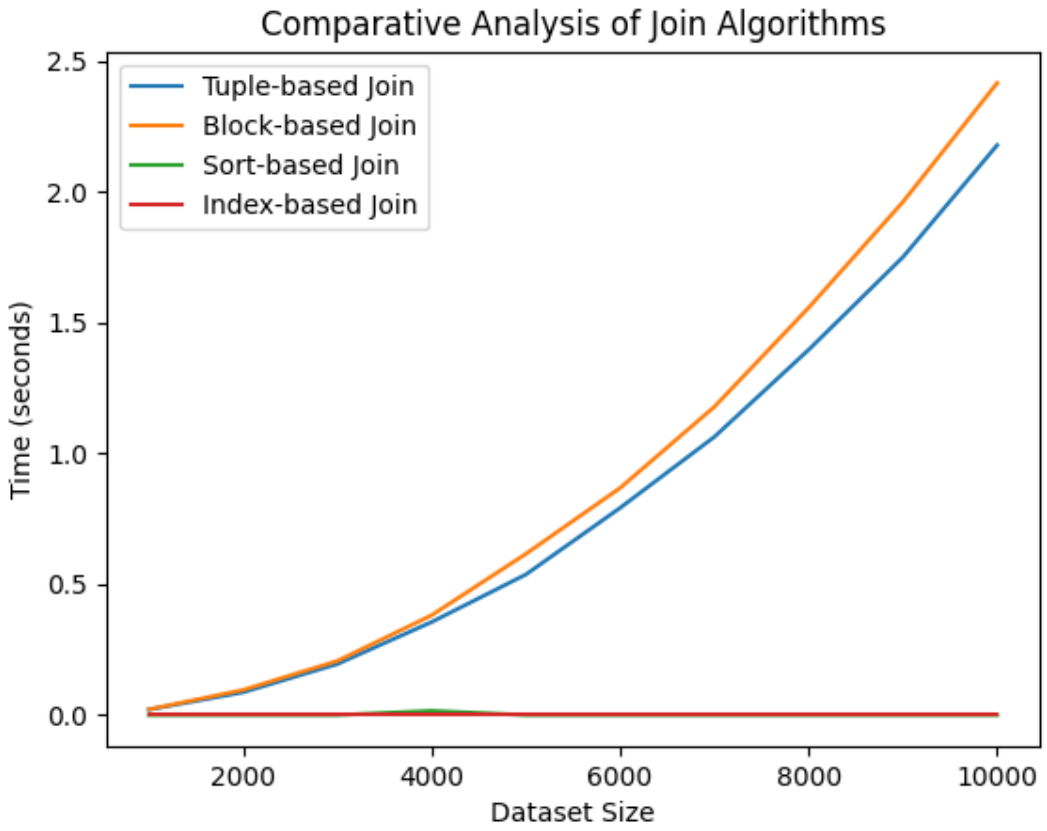
```
import time
import random
import matplotlib.pyplot as plt

# Tuple-based Join Algorithm
def tuple_join(A, B):
    result = []
    for a in A:
        for b in B:
            if a[0] == b[0]:
                result.append(a + b)
    return result

# Block-based Join Algorithm
def block_join(A, B, block_size):
    result = []
    for i in range(0, len(A), block_size):
        for j in range(0, len(B), block_size):
            for a in A[i:i+block_size]:
                for b in B[j:j+block_size]:
                    if a[0] == b[0]:
                        result.append(a + b)
    return result

# Sort-based Join Algorithm
def sort_join(A, B):
    A.sort(key=lambda x: x[0])
    B.sort(key=lambda x: x[0])
    result = []
    i = j = 0
    while i < len(A) and j < len(B):
        if A[i][0] < B[j][0]:
            i += 1
        elif A[i][0] > B[j][0]:
            j += 1
        else:
            result.append(A[i] + B[j])
            j += 1
    return result

# Index-based Join Algorithm
def index_join(A, B):
    index = {}
    for b in B:
```

```
        index[b[0]] = b
    result = []
    for a in A:
        if a[0] in index:
            result.append(a + index[a[0]])
    return result

# Function to generate synthetic datasets
def generate_dataset(size):
    return [(i, random.randint(1, 100)) for i in range(size)]

# Function to perform time analysis
def time_analysis(algorithm, dataset_sizes):
    times = []
    for size in dataset_sizes:
        A = generate_dataset(size)
        B = generate_dataset(size)
        start_time = time.time()
        algorithm(A, B)
        end_time = time.time()
        times.append(end_time - start_time)
    return times

# Dataset sizes to be tested
dataset_sizes = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

# Perform time analysis for each algorithm
tuple_times = time_analysis(tuple_join, dataset_sizes)
block_times = time_analysis(lambda A, B: block_join(A, B, 100),
dataset_sizes)  # Using block size of 100
sort_times = time_analysis(sort_join, dataset_sizes)
index_times = time_analysis(index_join, dataset_sizes)

# Plotting results for all join algorithms in the same window
fig, axs = plt.subplots(2, 2, figsize=(12, 8))

# Tuple-based Join
axs[0, 0].plot(dataset_sizes, tuple_times, label='Tuple-based Join')
axs[0, 0].set_title('Tuple-based Join')
axs[0, 0].set_xlabel('Dataset Size')
axs[0, 0].set_ylabel('Time (seconds)')

# Block-based Join
axs[0, 1].plot(dataset_sizes, block_times, label='Block-based Join')
axs[0, 1].set_title('Block-based Join')
axs[0, 1].set_xlabel('Dataset Size')
axs[0, 1].set_ylabel('Time (seconds)')
```

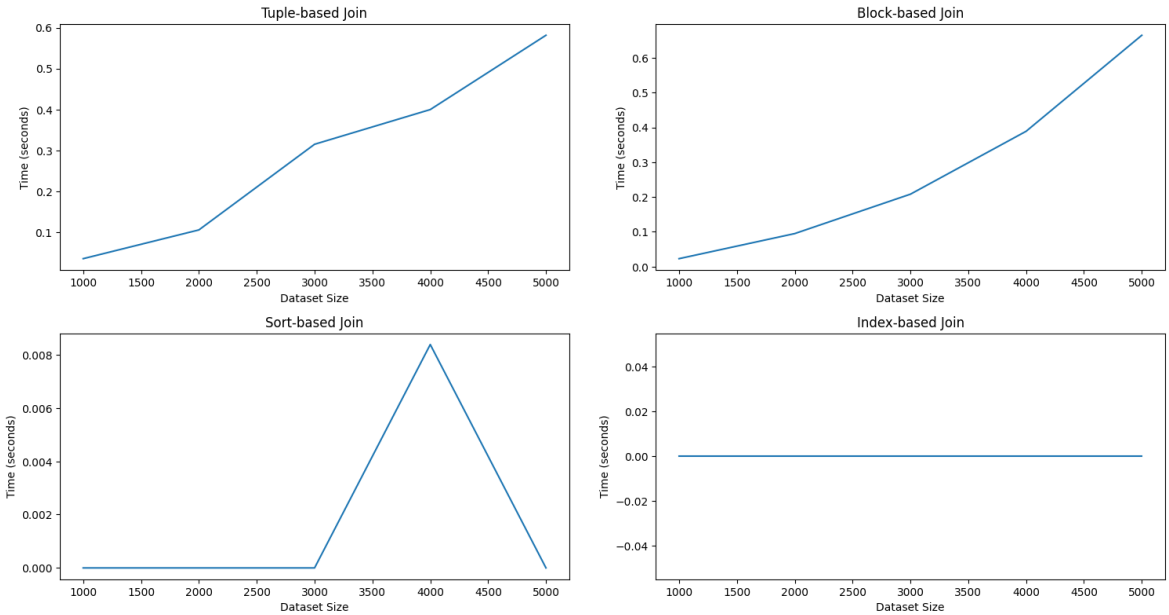
```
# Sort-based Join
axs[1, 0].plot(dataset_sizes, sort_times, label='Sort-based Join')
axs[1, 0].set_title('Sort-based Join')
axs[1, 0].set_xlabel('Dataset Size')
axs[1, 0].set_ylabel('Time (seconds)')

# Index-based Join
axs[1, 1].plot(dataset_sizes, index_times, label='Index-based Join')
axs[1, 1].set_title('Index-based Join')
axs[1, 1].set_xlabel('Dataset Size')
axs[1, 1].set_ylabel('Time (seconds)')

plt.tight_layout()
plt.show()
```

Results and Discussion:-

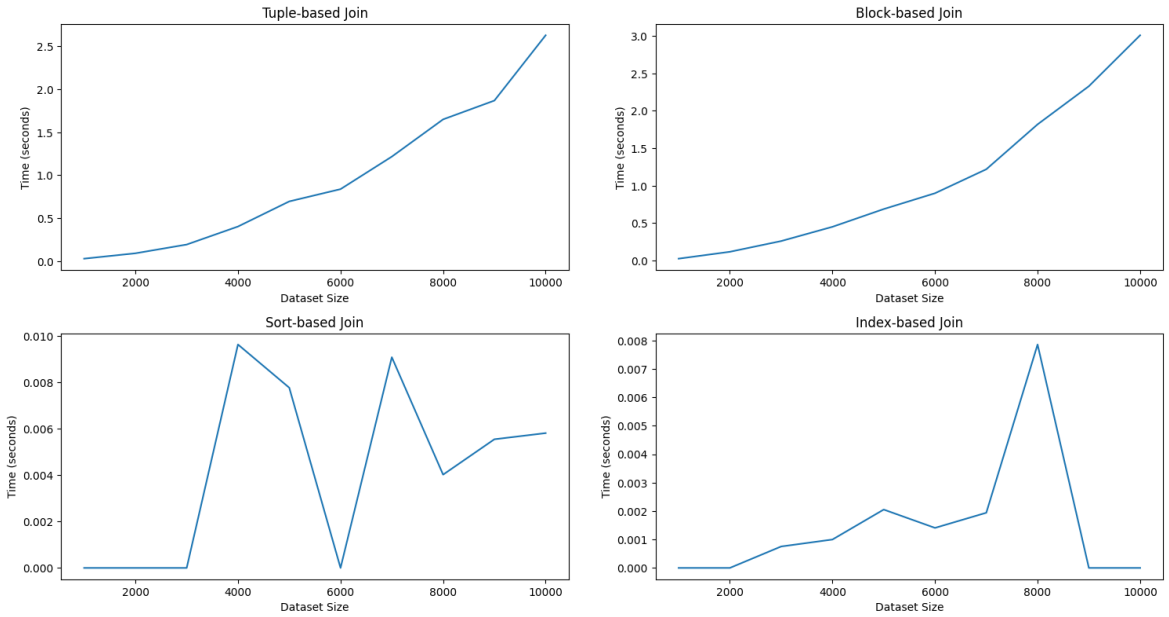




Output from dataset 1 – 1000 then 1 – 2000 and … till 1- 5000



Output from dataset 1 – 1000 then 1 – 2000 and … till 1- 10000



Conclusion:-

- **Tuple-based Join:-**
  - This algorithm has a straightforward implementation.
  - However, its performance degrades significantly as the dataset size increases, especially with large datasets, due to its quadratic time complexity.
- **Block-based Join:-**
  - Block-based join improves performance by reducing the number of disk I/O operations through block-wise processing.
  - It exhibits better scalability compared to tuple-based join but may still suffer from high overhead, especially with larger block sizes or uneven data distributions.
- **Sort-based Join:-**
  - Sort-based join achieves efficient join operations by sorting both input datasets and merging them in a single pass.
  - It offers better performance than tuple-based and block-based joins for moderately sized datasets, especially when datasets are pre-sorted or have similar distributions.
- **Index-based Join:-**
  - Index-based join utilizes indexing structures (e.g., hash tables or B-trees) to efficiently locate matching records.

- It provides fast access to data and can significantly reduce the computational cost, especially for large datasets, making it suitable for scenarios where indexing is feasible.

In conclusion, there is no one-size-fits-all "best" algorithm for join operations. Instead, the selection of the most suitable algorithm depends on the specific characteristics of the dataset and the query requirements. For small to moderately sized datasets, sort-based join may offer a good balance between efficiency and scalability. However, for large datasets or when indexed attributes are available, index-based join may outperform other algorithms due to its fast access to data. Ultimately, the choice of the "best" algorithm should be based on careful consideration of the dataset characteristics, computational resources, and performance requirements.

References:-

1. “Modern Database Systems" by Garcia-Molina, Ullman, and Widom.
2. "Database Management Systems" by Ramakrishnan and Gehrke.