《数据科学与工程算法》项目报告

报告题目：数据流统计算法

姓　　名：李芳

学　　号：10214602404

完成日期：2024.05.14

摘要 [中文]：

本次实验实现数据流统计算法，目的在于设计一个高效的影评数据流处理系统，来对现今互联网平台指数级影评数据进行分析与查询。实验在 Count-Min Sketch 和 Misra-Gries 两种数据结构相结合的基础上，实现了快速高效的成员查询、频率查询和 Top-k 查询功能。同时，对实现查询功能的两种数据结构的关键性能指标，例如查询精度，构建与更新时间，查询时间和空间占用等进行测试，以验证该影评数据流查询功能的高效性和准确性。本次实验为随时间增量式获取的影评数据流进行处理和查询提供了一种有效的解决方案，未来可以尝试不同优化方式来提高代码的执行效率。

Abstract [English]:

This experiment aims to design an efficient data stream processing system for analyzing and querying exponentially growing movie review data on contemporary internet platforms. Building upon the integration of Count-Min Sketch and Misra-Gries data structures, the experiment implements rapid and efficient member query, frequency query, and Top-k query functionalities. Additionally, key performance metrics of the two data structures used for query functionalities, such as query accuracy, construction and update time, query time, and space usage, are tested to validate the efficiency and accuracy of the movie review data stream querying functionalities. This experiment provides an effective solution for processing and querying movie review data streams acquired

incrementally over time.Future endeavors may explore avenues to enhance the efficiency of data stream processing.

一、 项目概述

1. 引言

随着现如今影音娱乐应用、论坛以及互联网的发展，人们无时无刻不产生着大量数据，如何收集、处理、应用数据是相关行业人员想方设法攻克的难题，其中也包含着电影媒体产业。在[MovieRens]这一电影推荐平台上，用户对电影的评分信息不断产生，形成庞大的数据流。为了更好地理解用户对电影的喜好、行为趋势以及市场需求，实现个性化推荐等功能，对这些指数级评分数据进行实时的分析和查询显得尤为重要。本次实验旨在设计并实现空间占用尽量小的数据结构，用于实现成员查询、频度查询和 Top-k 查询等功能，从而有效地分析和查询随时间增量式获取的电影评分数据流。

2. 科学价值与相关研究工作

- 数据流处理：数据流处理问题是现今大数据领域的研究热点之一。随着数据量的不断增加和数据产生速度的加快，如何在数据流中进行高效的查询和分析成为了亟待解决的问题。
- 空间效率和查询效率：如何设计空间占用尽量小的数据结构，如何实现快速高效的查询功能，这是提高数据处理的效率和性能至关重要的课题。相关研究工作主要集中在哈希技术、频率估计算法等方面。
- 用户行为分析与个性化推荐：[MovieRens]的电影评分数据包含丰富的用户行为信息，通过对这些数据的分析和挖掘，可以实现个性化推荐系统，提高推荐的精度和效果。

3. 项目主要内容

- 设计与实现数据结构： 在本次实验中，我设计并实现了两种主要的数据

结构，即 Count-Min Sketch 和 Misra-Gries。Count-Min Sketch 用于频度查询和成员查询功能，Misra-Gries 用于实现 Top-k 查询功能。这两种数据结构通过不同的原理和方法实现了对电影评分数据流的高效处理和查询。

- 功能实现：在两种数据结构基础上，我实现了成员查询（member_query）、频度查询（frequency_query）和 Top-k 查询（top_k_query）等功能。成员查询功能用于判断指定电影在给定时间戳之前是否被评分过；频度查询功能用于统计指定电影在给定时间戳之前被评分的总次数；Top-k 查询功能用于查询在给定时间戳之前被评分次数最多的前 k 个电影。
- 性能指标展示：在实验中，随机产生 100 对电影 ID 和时间戳进行测评，在报告中，我选取典型测试数据展示数据结构的关键性能指标，包括查询精度、构建与更新时间、查询时间和空间占用等。这些指标将有助于评估和比较不同数据结构在处理电影评分数据流时的性能优劣，为进一步优化和改进提供参考和依据。

通过以上内容的设计和实现，来构建一个高效、灵活且功能丰富的数据处理系统，以满足对电影评分数据流进行实时分析和查询的需求。同时，后部分展示的数据结构的关键性能指标，将有助于评估其在实际应用中的表现和可行性，为相关研究和实践提供有益的参考和启示。

二、 问题定义

在 ratings.csv 文件中，每行评分都包含 userId,movieId,rating,timestamp 信息，形成了一个随时间逐渐增量的数据流。

实验目的是设计高效的数据结构，能够以尽可能小的空间开销来处理这个数据流，并实现以下功能：

1. 成员查询（member_query）：

(1) 语言描述：对于给定的查询时间戳 timestamp 和 movieID，判断 ratings.csv 是否存在 time≤timestamp 的时间点，该用户对电影 movie 进行了评分，即该用户此时间之前该用户对该电影是否存在评分记录。

(2) 数学形式：

$$\text{member\_query}(\text{movieID}, \text{timestamp})$$
$$= \begin{cases} \text{True}, & \text{if} \exists (\text{movie}, \text{time}), \quad \text{such that time} \leq \text{timestamp} \\ \text{False}, & \text{otherwise} \end{cases}$$

2. 频度查询（frequency_query）：

(1) 语言描述：对于给定的查询时间戳 timestamp 和 movieID，统计 ratings.csv 在时间戳 time≤timestamp 的时间段内，电影 movie 被评分的总次数。

(2) 数学形式：

$$\text{frequency\_query}(\text{movieID}, \text{timestamp})$$
$$= \sum \text{time} \leq \text{timestamp}\, \delta(\text{movie}, \text{time}),$$
$$\text{其中} \delta(\text{movie}, \text{time}) == \begin{cases} 1, & \text{if} \exists (\text{movie}, \text{time}) \text{ is rated} \\ 0, & \text{otherwise} \end{cases}$$

2. Top-k 查询（top_k_query）：

(1) 语言描述：对于给定正整数 k 和查询时间戳 timestamp，找出 ratings.csv 在时间戳 time≤timestamp 的时间段内，被评分次数最多的前 k 个电影。

(2) 数学形式：

$$\text{top\_k\_query}(k, \text{timestamp}) = \text{argmax}_{\text{moive} \in \text{movieID}} \text{count}(\text{movie}, \text{time}),$$

其中，$\text{count}(\text{movie}, \text{time})$代表时间戳 time $\leq$ timestamp 时，movie 的评分次数

三、 方法

1. 问题解决步骤：

本次实验需要解决的主要问题是设计并实现空间占用尽量小的数据结构，以支持成员查询、频度查询和 Top-k 查询功能。步骤如下：

(1) 数据结构设计：

设计了两种主要的数据结构来解决问题：Count-Min Sketch 和 Misra-Gries 算法：

Count-Min Sketch 是一种概率型数据结构，用于频率估计。它通过一个二维数组来存储频率信息，使用多个哈希函数将元素映射到计数器矩阵中，并利用增量更新实现对数据流的实时统计，广泛应用于网络流量监测、实时计数等领域。

Misra-Gries 算法是一种经典的频率估计算法，适用于有限的空间内实现频率统计。该算法使用固定大小的字典存储频率信息，保留前 k 个高频元素的计数器，通过逐步减少计数器值来实现对高频元素的统计，并能够动态调整计数器值以适应数据流的变化，常用于实时数据流处理、频率监控等场景。

(2) 功能实现：

在确定数据结构后，根据实验任务的描述，实现了成员查询、频度查询和 Top-k 查询功能。这些功能分别对应了对数据结构的不同操作，如增量更新、频率估计和排序选择。

(3) 性能评估：

最后，对设计的数据结构和功能进行性能评估，包括查询精度、构建与更新时间、查询时间和空间占用等指标的评估，实现字典的三种查询功能作为正确答案基准，以验证其在处理随时间增量式电影评分数据集时的有效性和效率。

2. 详细实现细节：

(1) Count-Min Sketch：

① 初始化：Count-Min Sketch 初始化为指定初始宽度（initial_width）和深度（depth），以及填充比例阈值（fill_ratio_threshold）。

```
class CountMinSketch:
    def __init__(self, initial_width, depth, fill_ratio_threshold=0.8):
        self.width = initial_width
        self.depth = depth
        self.counters = np.zeros( shape: (depth, initial_width), dtype=int)
        self.fill_ratio_threshold = fill_ratio_threshold
```

② 增量更新：使用depth个哈希函数将元素映射到计数器矩阵的特定位置，并将对应位置的计数器值加一。在填充比例超过阈值时，进行扩容操作。

```
def increment(self, key):
    if self.fill_ratio() > self.fill_ratio_threshold:
        self.expand()
    for i in range(self.depth):
        hash_val = hash(str(i) + key) % self.width
        self.counters[i, hash_val] += 1
```

③ 频率估计：对于给定的元素，使用 depth 个哈希函数计算其在计数器矩阵中的位置，并返回所有位置上的最小计数器值作为频率估计结果。

```
def estimate(self, key):
    min_count = float('inf')
    for i in range(self.depth):
        hash_val = hash(str(i) + key) % self.width
        min_count = min(min_count, self.counters[i, hash_val])
    return min_count
```

④ 扩容策略：当计数器矩阵的填充比例超过设定的阈值时，进行矩阵宽度的扩容操作，以保持较低的误差率。

1) 计算填充比率：计算 Count-Min Sketch 数据结构中已使用的计数器占总计数器数的比例。通过填充比率，可以确定何时需要进行扩展以增加数据结构的容量。

```
def fill_ratio(self):
    return np.count_nonzero(self.counters) / (self.width * self.depth)
```

2) 扩展：在填充比率超过阈值时，扩展 Count-Min Sketch 数据结构的宽度，以容纳更多的计数器。先将当前计数器矩阵复制到新的更宽的矩阵中，再更新数据结构的宽度和计数器矩阵。

```
def expand(self):
    new_width = self.width * 2
    new_counters = np.zeros( shape: (self.depth, new_width), dtype=int)
    for i in range(self.depth):
        for j in range(self.width):
            new_counters[i, j] = self.counters[i, j]
    self.width = new_width
    self.counters = new_counters
```

(2) Misra-Gries 算法：

① 初始化：Misra-Gries 算法初始化时需要指定初始 k 值，用于确定保留的高频元素数量。

```
class MisraGries:
    def __init__(self, initial_k):
        self.initial_k = initial_k
        self.k = 50 * self.initial_k
        self.items = {}
```

② 增量更新：对于每个新的元素，首先判断是否在已保存的计数器中，若

是则加一，否则检查当前保存的计数器数量是否小于 k，若是则将其加入保存的计数器中，若不是则逐步减少已保存的计数器值，当频度值减少到 0 的时候去除该元素。

```
def process_item(self, item):
    if item in self.items:
        self.items[item] += 1
    elif len(self.items) < self.k - 1:
        self.items[item] = 1
    else:
        for key in list(self.items.keys()):
            self.items[key] -= 1
            if self.items[key] == 0:
                del self.items[key]
        self.items[item] = self._calculate_frequency()
```

③ 频率估计：返回保存的前 k 个高频元素及其对应的计数器值作为频率估计结果。

```
def get_top_k(self):
    return sorted(self.items.items(), key=lambda x: x[1], reverse=True)[:self.initial_k]
```

④ 频率调整：根据当前保存的计数器数量动态调整计数器值，减少与真实技术频度值的误差。

```
def _calculate_frequency(self):
    if len(self.items) < self.k // 2:
        return max(1, self.initial_k // 10)
    elif len(self.items) < self.k:
        return max(1, self.initial_k // 5)
    else:
        return max(1, self.initial_k // 2)
```

(3) 查询功能：**Query.py** 代码实现查询功能

① 成员查询（member_query）：

1) 时间复杂度：O(d)，其中 d 是 CountMinSketch 的深度 depth。

2) 首先创建一个 Count-Min Sketch 数据结构。接着从文件中读取评分数据，对于每个评分记录，如果其时间戳早于或等于查询时间戳，则将相应的电影标识符增加到 Count-Min Sketch 中。最后，查询

给定的电影标识符是否存在于 Count-Min Sketch 中。如果存在，则返回 True，否则返回 False。

```
def member_query(movieId, timestamp, initial_width, depth, fill_ratio_threshold=0.8):
    count_min_sketch = CountMinSketch(initial_width, depth, fill_ratio_threshold)
    with open('ratings.csv', 'r') as file:
        next(file)
        for line in file:
            _, movie, _, time = line.strip().split(',')
            if int(time) <= timestamp:
                count_min_sketch.increment(movie)

    flag = False
    if count_min_sketch.estimate(movieId) > 0:
        flag = True
    return flag
```

② 频度查询（frequency_query）：

1) 时间复杂度：O(d)，其中 d 是 CountMinSketch 的深度 depth。

2) 首先创建一个 Count-Min Sketch 数据结构。然后遍历评分数据，将每个电影标识符的出现次数添加到 Count-Min Sketch 中。最后，查询给定的电影标识符在 Count-Min Sketch 中的计数。如果计数为 'inf'，则表示计数大于等于 Count-Min Sketch 的宽度，即频度超出了所能表示的范围，此时将计数设置为 0。

```
def frequency_query(movieId, timestamp, initial_width, depth, fill_ratio_threshold=0.8):
    count_min_sketch = CountMinSketch(initial_width, depth, fill_ratio_threshold)
    with open('ratings.csv', 'r') as file:
        next(file)
        for line in file:
            _, movie, _, time = line.strip().split(',')
            if int(time) <= timestamp:
                count_min_sketch.increment(movie)

    freq = count_min_sketch.estimate(movieId)
    if freq == 'inf':
        freq = 0
    return freq
```

③ Top-k 查询（top_k_query）：

1) 时间复杂度：O(n)，其中 n 是数据文件中的记录数。

2) 首先，创建一个 MisraGries 对象，初始化时指定 k 值。然后，遍历评分数据，对于每个电影标识符，将其传递给 MisraGries 对象

的 process_item 方法。最后，调用 MisraGries 对象的 get_top_k 方法获取频率最高的前 k 个电影标识符。

```
def top_k_query(k, timestamp):
    misra_gries = MisraGries(k)
    with open('ratings.csv', 'r') as file:
        next(file)
        for line in file:
            _, movie, _, time = line.strip().split(',')
            if int(time) <= timestamp:
                misra_gries.process_item(movie)
    top_k_items = misra_gries.get_top_k()
    return top_k_items
```

④ main 函数：命令行选择 1-3 来对应选择成员查询、频度查询、topk 查询。整数 1，2 时会要求输入电影 ID 和时间戳，整数 3 时会要求输入 k 和时间戳。查询结束后，继续询问用户是否继续查询。构建初始数值由数据量大小而定，以下举例：

```
def main():
    while True:
        print("请选择查询功能：\n1. 成员查询\n2. 频度查询\n3. Top-k 查询")
        choice = input("请输入查询功能选项 (1-3): ")

        if choice not in ['1', '2', '3']:
            print("请选择有效的选项 (1-3)！")
            continue
        if choice == '1' or choice == '2':
            movieId = input("请输入电影 ID: ")
            timestamp = int(input("请输入时间戳: "))
            if choice == '1':
                result = member_query(movieId, timestamp, initial_width=10000, depth=10)
                if result:
                    print(f"电影 {movieId} 在时间戳 {timestamp} 及之前被评分过。")
                else:
                    print(f"电影 {movieId} 在时间戳 {timestamp} 及之前未被评分过。")
            elif choice == '2':
                frequency = frequency_query(movieId, timestamp, initial_width=10000, depth=10)
                print(f"电影 {movieId} 在时间戳 {timestamp} 及之前被评分的总次数为: {frequency}")
        elif choice == '3':
            k = int(input("请输入要查询的前 k 个电影数目: "))
            timestamp = int(input("请输入时间戳: "))
            top_k_items = top_k_query(k, timestamp)
            print(f"时间戳 {timestamp} 及之前被评分次数最多的前 {k} 个电影:")
            for i, (movie, count) in enumerate(top_k_items, start=1):
                print(f"{i}. 电影 ID: {movie}, 评分次数: {count}")

        another_query = input("是否继续进行其他查询？(yes/no): ")
        if another_query.lower() != 'yes':
            break
```

(4) 性能评估：

对设计的数据结构和查询功能进行综合性能评估的代码实现在 test.py 文件中，主要思路是：

先随机生成 **100 对**电影 ID 和时间戳，写入 query.csv 文件中：

```
#生成测试用的100条数据文件
if __name__ == "__main__":
    num_queries = 100
    file_path = 'query.csv'
    generate_random_data(file_path, num_queries)
    print(f"{num_queries} pairs of (movieID, timestamp) have been generated and saved to '{file_path}'.")
```

然后对该文件每行进行查询精度的测试查询和字典正确查询，并将结果保存在新 csv 文件中，供后面评估用，下面以 member 的测试代码为例：

```
#member
if __name__ == "__main__":
    with open('query.csv', 'r') as file:
        next(file)
        with open('member_test.csv', 'w', newline='') as csvfile:
            fieldnames = ['result', 'construction_time', 'update_time', 'query_time', 'space_occupancy', 'true_result']
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
            writer.writeheader()
            for line in file:
                movieId, timestamp = line.strip().split(',')
                result = member_query(movieId, int(timestamp), initial_width=10000, depth=10)
                true_result = dic_member_query(int(movieId), int(timestamp))
                writer.writerow({
                    'result': result[0],
                    'construction_time': result[1],
                    'update_time': result[2],
                    'query_time': result[3],
                    'space_occupancy': result[4],
                    'true_result': true_result
                })
```

将两种 query 后的结果标签作为文件列索引：测试查询结果、构建时间、元素更新时间、查询时间、字典正确结果，输出到 member_test.csv 文件中。

对所实现的查询功能，有如下评估指标：

① 查询精度：对于三种查询的查询精度测评，以字典进行查询功能的结果为基准，进行结果对比，来评估查询结果的准确性：

1) 字典的成员查询功能：

时间复杂度 O(n)，其中 n 是数据文件中的记录数。

```python
def dic_member_query(movieId, timestamp):
    data_dict = {}
    with open('ratings.csv', 'r') as file:
        next(file)
        for line in file:
            _, movie, _, timesta = line.strip().split(',')
            if int(timesta) <= timestamp:
                data_dict[movie] = data_dict.get(movie, 0) + 1
    return movieId in data_dict
```

2) 字典的频度查询功能：

时间复杂度：O(n)，其中 n 是数据文件中的记录数。

```python
def dic_frequency_query(movieId, timestamp):
    data_dict = {}
    with open('ratings.csv', 'r') as file:
        next(file)
        for line in file:
            _, movie, _, timesta = line.strip().split(',')
            if int(timesta) <= timestamp:
                data_dict[movie] = data_dict.get(movie, 0) + 1
    return data_dict.get(movieId, 0)
```

3) 字典的 topk 查询功能：

时间复杂度：O(n*log(n))，其中 n 是数据文件中的记录数。

```python
def dic_top_k_query(k, timestamp):
    data_dict = {}
    with open('ratings.csv', 'r') as file:
        next(file)
        for line in file:
            _, movie, _, timesta = line.strip().split(',')
            if int(timesta) <= timestamp:
                data_dict[movie] = data_dict.get(movie, 0) + 1
    return sorted(data_dict.items(), key=lambda x: x[1], reverse=True)[:k]
```

② 构建与更新时间：构建两种数据结构初始化所需时间以及每次增量更新的时间总和除以增量总个数的平均更新时间开销。

1) 构建时间测量：三种查询功能该部分测量方法大致相同，都是测量读取 ratings.csv 文件小于给定时间戳 timestamp 的数据加入到两

种数据结构，这一步骤完成的耗费时间。

以 member_query 的构建时间测量部分为例，如下图：

```
construction_start = time.time()
count_min_sketch = CountMinSketch(initial_width, depth, fill_ratio_threshold)
with open('ratings.csv', 'r') as file:
    next(file)
    for line in file:
        _, movie, _, timesta = line.strip().split(',')
        if int(timesta) <= timestamp:
            update_time += count_min_sketch.increment(movie)
            incre_count += 1
construction_end = time.time()
```

2) 更新时间测量：三种查询的该部分测量方法大致相同，都是在两种数据结构进行增量的函数中头尾测量更新时间，并返回后者减去前者的差值，避免不好测量的情况，我在 query 函数中使用了 incre_count 来记录总更新元素数量，返回总更新时间/incre_count 这一平均更新时间。

a. Count_min_sketch 中增量函数单个元素增加时的更新时间测量如下：

```
def increment(self, key):
    updata_start = time.time()
    if self.fill_ratio() > self.fill_ratio_threshold:
        self.expand()
    for i in range(self.depth):
        hash_val = hash(str(i) + key) % self.width
        self.counters[i, hash_val] += 1
    updata_end = time.time()
    return updata_end - updata_start
```

b. Misra_gries 中增量函数单个元素增加时的更新时间测量如下：

```
def process_item(self, item):
    updata_start = time.time()
    if item in self.items:
        self.items[item] += 1
    elif len(self.items) < self.k - 1:
        self.items[item] = 1
    else:
        for key in list(self.items.keys()):
            self.items[key] -= 1
            if self.items[key] == 0:
                del self.items[key]
        self.items[item] = self._calculate_frequency()
    updata_end = time.time()
    return updata_end - updata_start
```

③ 查询时间：测试每次成员查询、频度查询和 Top-k 查询的查询部分耗费时间。

1) Member_query:

```
query_start = time.time()
flag = False
if count_min_sketch.estimate(movieId) > 0:
    flag = True
query_end = time.time()
```

2) Frequency_query:

```
query_start = time.time()
freq = count_min_sketch.estimate(movieId)
if freq == 'inf':
    freq = 0
query_end = time.time()
```

3) Top_k_query:

```
query_start = time.time()
top_k_items = misra_gries.get_top_k()
query_end = time.time()
```

④ 空间占用：使用 sys.getsizeof()获取数据结构在内存中的实时空间占用情况，包括 Count-Min Sketch 的计数器矩阵、一些参数以及 Misra-Gries 算法的保存计数器、一些参数总共占用空间大小。

1) Count-Min Sketch：

```
space_occupancy = sys.getsizeof(count_min_sketch.counters.nbytes) + sys.getsizeof(
    count_min_sketch.width) + sys.getsizeof(count_min_sketch.depth) + sys.getsizeof(
    count_min_sketch.fill_ratio_threshold)
```

2) Misra-Gries：

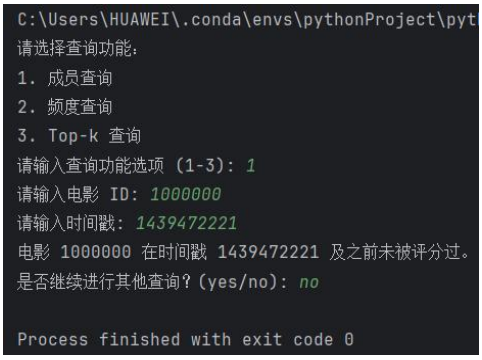
```
space_occupancy = sys.getsizeof(misra_gries.k) + sys.getsizeof(misra_gries.initial_k) + sys.getsizeof(
    misra_gries.items)
```

通过对这些性能指标的评估，我们可以全面地了解所设计的数据结构和功能在处理电影评分数据集时的表现和效果，为进一步优化奠定基础。

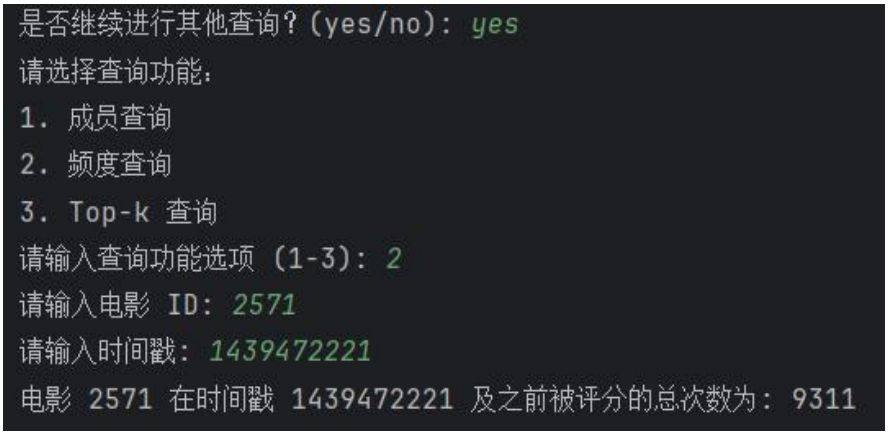
四、 实验结果

1. **query.py** 中实现了三种查询功能，可以通过命令行输入的参数进行查询功能选择，以及参数读取等功能，结果如下：

(1) 成员查询 member_query:



(2) 频度查询 frequency_query:



(3) Topk 查询 top_k_query:

```
是否继续进行其他查询？(yes/no): yes
请选择查询功能：
1. 成员查询
2. 频度查询
3. Top-k 查询
请输入查询功能选项 (1-3): 3
请输入要查询的前 k 个电影数目: 10
请输入时间戳: 1573945484
时间戳 1573945484 及之前被评分次数最多的前 10 个电影:
1. 电影 ID: 356, 评分次数: 31902
2. 电影 ID: 318, 评分次数: 31880
3. 电影 ID: 296, 评分次数: 30087
4. 电影 ID: 593, 评分次数: 24546
5. 电影 ID: 2571, 评分次数: 23083
6. 电影 ID: 260, 评分次数: 19129
7. 电影 ID: 480, 评分次数: 14583
8. 电影 ID: 527, 评分次数: 10834
9. 电影 ID: 110, 评分次数: 9617
10. 电影 ID: 2959, 评分次数: 9177
是否继续进行其他查询？(yes/no): no
```

2. **test.py** 文件中采用 100 对随机生成的电影 ID 和时间戳对三种查询功能进行了多维度性能评估，下面以功能为区分，选取特定有区分度的数据，进行详细的测试结果展示：

(1) 成员查询 member_query：

共选取八个典型样本，时间跨越，电影 ID 跨越大，True&False 都有

```
成员查询测试：
电影ID：1260，时间戳：1147877857；测试答案：True
正确答案：True
本次测试指标：
此次成员查询时间：0.0
构建时间：368.9637954235077
平均更新元素时间：3.058478897250825e-05
空间占用：108
电影ID：2571，时间戳：1439472221；测试答案：True
正确答案：True
本次测试指标：
此次成员查询时间：0.0
构建时间：1040.3581569194794
平均更新元素时间：5.502971566979684e-05
空间占用：108
电影ID：96737，时间戳：1439474741；测试答案：True
正确答案：True
本次测试指标：
此次成员查询时间：0.0
构建时间：1019.7267215251923
平均更新元素时间：5.389638319457885e-05
空间占用：108
电影ID：63876，时间戳：1240952515；测试答案：True
正确答案：True
本次测试指标：
此次成员查询时间：0.0
构建时间：462.38520550727844
平均更新元素时间：3.08014451153202e-05
空间占用：108
```

```
电影ID：400000，时间戳：1016738289；测试答案：False
正确答案：False
本次测试指标：
此次成员查询时间：0.0
构建时间：227.4721999168396
平均更新元素时间：3.106328157992347e-05
空间占用：108
电影ID：1260，时间戳：10000000000；测试答案：True
正确答案：True
本次测试指标：
此次成员查询时间：0.0
构建时间：3282.0759420394897
平均更新元素时间：0.00012928683279527097
空间占用：108
电影ID：1，时间戳：830000000；测试答案：True
正确答案：True
本次测试指标：
此次成员查询时间：0.0
构建时间：23.15637516975403
平均更新元素时间：3.197934182502799e-05
空间占用：108
电影ID：2，时间戳：1600000000；测试答案：True
正确答案：True
本次测试指标：
此次成员查询时间：0.0
构建时间：3094.953247308731
平均更新元素时间：0.00012192795378421873
空间占用：108
```

- count-min sketch 在成员查询功能上查询精度很高，查询结果几乎不出现误差；
- 查询几乎不花时间，该结构在成员查询功能上表现比较好；构建时间随着时间戳增加而增加，但耗时均不低；
- 元素平均更新时间很小，但会稍有差异；
- 空间占用基本无差，因为初始宽度设置为 10000，深度设置为 10，足够存放，基本不需要扩张情况，否则，构建时间和空间占用均会增加。

(2) 频度查询 frequency_query：

选取四组电影 ID 和时间戳组合，电影 ID 以及时间戳跨越大，高频&低频均涉及。

```
频度查询测试：
电影ID：1260，时间戳：1147877857：测试答案：2442
正确答案：2442
本次测试指标：
此次频度查询时间：0.0019996166229248047
构建时间：367.60591983795166
平均更新元素时间：3.051716016848391e-05
空间占用：108
电影ID：2571，时间戳：1439472221：测试答案：9615
正确答案：48020
本次测试指标：
此次频度查询时间：0.0
构建时间：1030.2884929180145
平均更新元素时间：5.447734019801135e-05
空间占用：108
电影ID：96737，时间戳：1439474741：测试答案：153
正确答案：792
本次测试指标：
此次频度查询时间：0.0
构建时间：1029.8176198005676
平均更新元素时间：5.4452713265458306e-05
空间占用：108
电影ID：63876，时间戳：1240952515：测试答案：551
正确答案：551
本次测试指标：
此次频度查询时间：0.0
构建时间：467.4831190109253
平均更新元素时间：3.118903980125918e-05
空间占用：108
```

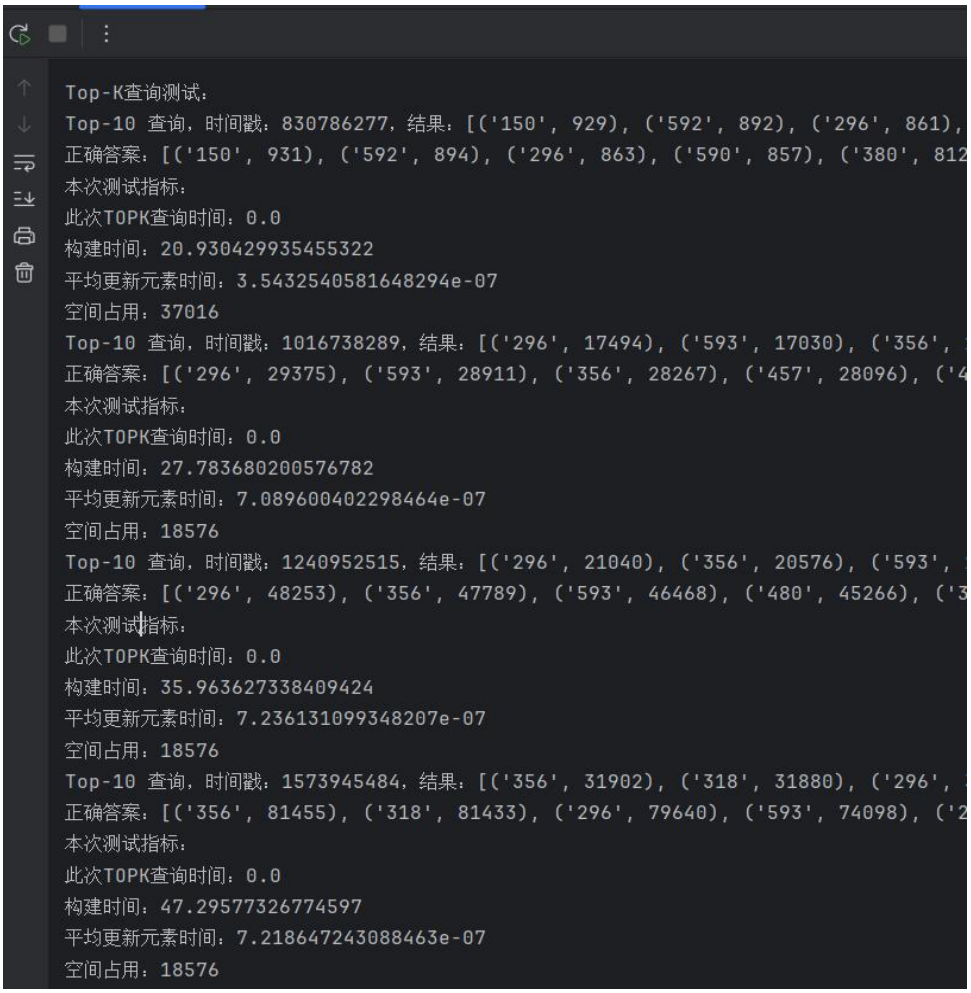
- count-min sketch 在频度查询功能上的精度与时间戳大小有很大关

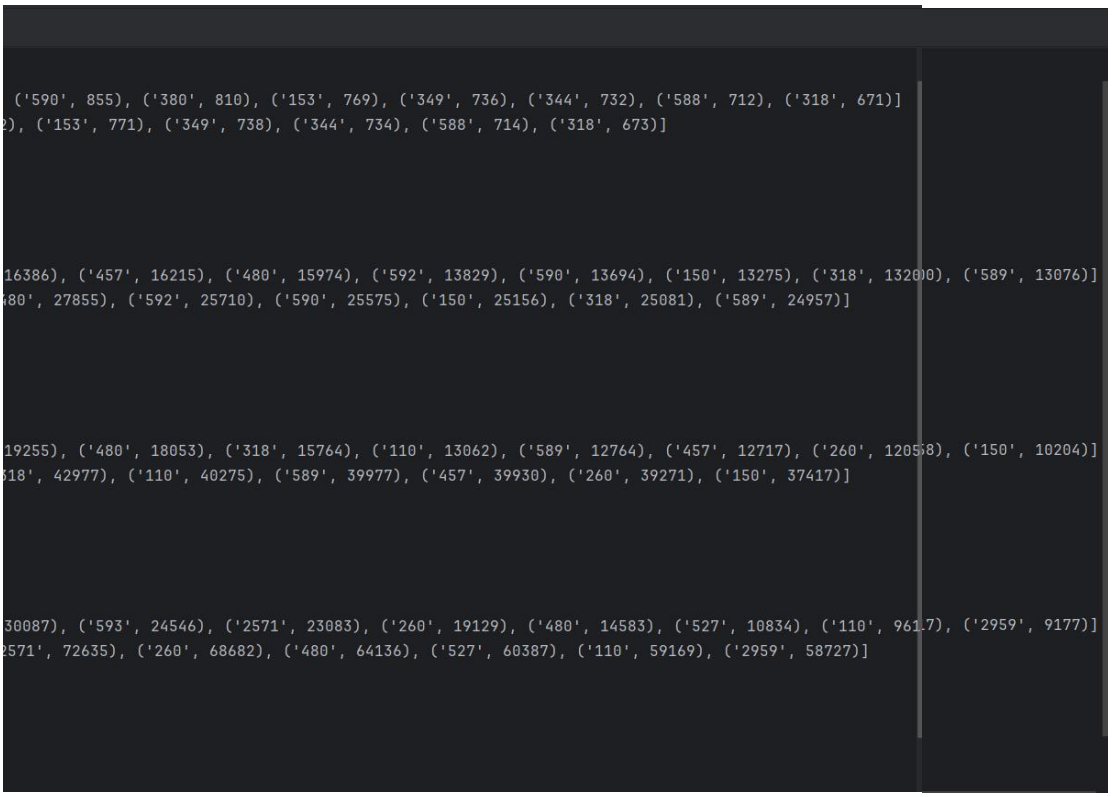
系，当时间戳特别大时，查询结果相比实际频度会出现较大偏差；

- 查询几乎不耗时；
- count-min sketch 的构建时间随时间戳增大而增加；
- 每个元素平均更新时间很小，稍有差异；
- 空间占用基本无差。

(3) Topk 查询 top_k_query：

选取四组时间戳分布较平均的实验测试数据，令 k=10，找出前十名电影 ID 及频度。

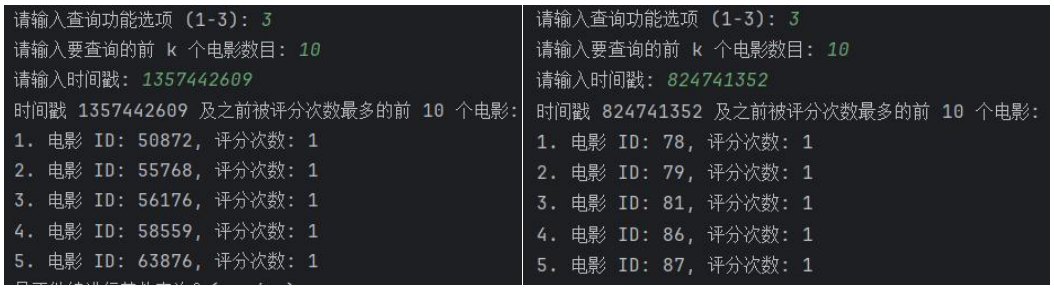




- Misra-Gries 在找到数据流中出现频率最高的前 k 个元素该功能上查询精度较高，但返回的频度估计值就与真实值有较大差别。
- 查询几乎不耗时；
- 构建时间相比 CMSketch 十分快速，且平均元素更新时间也较小，稍有差异；
- 空间占用会因为数据流中频率的不同而会有差别，此时设置 K=50*initial_k。

对于 TOPK 查询，我还进行了 k 选取的一些测试：

- k=initial_k:



- k=10*initial_k:

```
请输入查询功能选项 (1-3): 3
请输入要查询的前 k 个电影数目: 10
请输入时间戳: 1357442609
时间戳 1357442609 及之前被评分次数最多的前 10 个电影:
1. 电影 ID: 2968, 评分次数: 1
2. 电影 ID: 2997, 评分次数: 1
3. 电影 ID: 3039, 评分次数: 1
4. 电影 ID: 3052, 评分次数: 1
5. 电影 ID: 3105, 评分次数: 1
6. 电影 ID: 3107, 评分次数: 1
7. 电影 ID: 3155, 评分次数: 1
8. 电影 ID: 3173, 评分次数: 1
9. 电影 ID: 3179, 评分次数: 1
10. 电影 ID: 3255, 评分次数: 1
```

```
请输入查询功能选项 (1-3): 3
请输入要查询的前 k 个电影数目: 10
请输入时间戳: 824741352
时间戳 824741352 及之前被评分次数最多的前 10 个电影:
1. 电影 ID: 10, 评分次数: 11
2. 电影 ID: 21, 评分次数: 10
3. 电影 ID: 19, 评分次数: 7
4. 电影 ID: 50, 评分次数: 7
5. 电影 ID: 11, 评分次数: 7
6. 电影 ID: 17, 评分次数: 7
7. 电影 ID: 32, 评分次数: 7
8. 电影 ID: 47, 评分次数: 6
9. 电影 ID: 1, 评分次数: 6
10. 电影 ID: 6, 评分次数: 6
```

- k=50*initial_k:

```
请输入要查询的前 k 个电影数目: 10
请输入时间戳: 1357442609
时间戳 1357442609 及之前被评分次数最多的前 10 个电影:
1. 电影 ID: 296, 评分次数: 23335
2. 电影 ID: 356, 评分次数: 22737
3. 电影 ID: 593, 评分次数: 20150
4. 电影 ID: 480, 评分次数: 18520
5. 电影 ID: 318, 评分次数: 18516
6. 电影 ID: 110, 评分次数: 12973
7. 电影 ID: 260, 评分次数: 12522
8. 电影 ID: 589, 评分次数: 12028
9. 电影 ID: 457, 评分次数: 10439
10. 电影 ID: 1, 评分次数: 9382
```

```
请输入要查询的前 k 个电影数目: 10
请输入时间戳: 824741352
时间戳 824741352 及之前被评分次数最多的前 10 个电影:
1. 电影 ID: 10, 评分次数: 11
2. 电影 ID: 21, 评分次数: 10
3. 电影 ID: 19, 评分次数: 7
4. 电影 ID: 50, 评分次数: 7
5. 电影 ID: 11, 评分次数: 7
6. 电影 ID: 17, 评分次数: 7
7. 电影 ID: 32, 评分次数: 7
8. 电影 ID: 47, 评分次数: 6
9. 电影 ID: 1, 评分次数: 6
10. 电影 ID: 6, 评分次数: 6
```

- k=100*initial_k:

```
请输入要查询的前 k 个电影数目: 10
请输入时间戳: 1357442609
时间戳 1357442609 及之前被评分次数最多的前 10 个电影:
1. 电影 ID: 296, 评分次数: 41123
2. 电影 ID: 356, 评分次数: 40525
3. 电影 ID: 593, 评分次数: 37938
4. 电影 ID: 480, 评分次数: 36308
5. 电影 ID: 318, 评分次数: 36304
6. 电影 ID: 110, 评分次数: 30761
7. 电影 ID: 260, 评分次数: 30310
8. 电影 ID: 589, 评分次数: 29816
9. 电影 ID: 457, 评分次数: 28227
10. 电影 ID: 1, 评分次数: 27170
```

```
请输入要查询的前 k 个电影数目: 10
请输入时间戳: 824741352
时间戳 824741352 及之前被评分次数最多的前 10 个电影:
1. 电影 ID: 10, 评分次数: 11
2. 电影 ID: 21, 评分次数: 10
3. 电影 ID: 19, 评分次数: 7
4. 电影 ID: 50, 评分次数: 7
5. 电影 ID: 11, 评分次数: 7
6. 电影 ID: 17, 评分次数: 7
7. 电影 ID: 32, 评分次数: 7
8. 电影 ID: 47, 评分次数: 6
9. 电影 ID: 1, 评分次数: 6
10. 电影 ID: 6, 评分次数: 6
```

- 字典返回的正确答案:

```
请输入要查询的前 k 个电影数目: 10
请输入时间戳: 1357442609
时间戳 1357442609 及之前被评分次数最多的前 10 个电影:
1. 电影 ID: 296, 评分次数: 55784
2. 电影 ID: 356, 评分次数: 55186
3. 电影 ID: 593, 评分次数: 52599
4. 电影 ID: 480, 评分次数: 50969
5. 电影 ID: 318, 评分次数: 50965
6. 电影 ID: 110, 评分次数: 45422
7. 电影 ID: 260, 评分次数: 44971
8. 电影 ID: 589, 评分次数: 44477
9. 电影 ID: 457, 评分次数: 42888
10. 电影 ID: 1, 评分次数: 41831
```

```
请输入要查询的前 k 个电影数目: 10
请输入时间戳: 824741352
时间戳 824741352 及之前被评分次数最多的前 10 个电影:
1. 电影 ID: 10, 评分次数: 11
2. 电影 ID: 21, 评分次数: 10
3. 电影 ID: 19, 评分次数: 7
4. 电影 ID: 50, 评分次数: 7
5. 电影 ID: 11, 评分次数: 7
6. 电影 ID: 17, 评分次数: 7
7. 电影 ID: 32, 评分次数: 7
8. 电影 ID: 47, 评分次数: 6
9. 电影 ID: 1, 评分次数: 6
10. 电影 ID: 6, 评分次数: 6
```

由上可知，Misra-Gries 算法的查询精度与存储数组大小密切相关，过小会导致查询无效；使查询结果有效的一定大小基础上，k 值越大，越准确，对应的耗时和空间占用也会对应增加。

五、 结论

1. 不足与分析：

(1) 空间占用：

在这次实验中，为了实现数据流统计算法，我使用了 Count-Min Sketch 和 Misra-Gries 这两种数据结构，这两种算法在具体实行查询功能时具有较高的效率，但是也可以从测试结果中得出，要想获得更高的查询精度，必然要牺牲一些空间优势，需要把它们的空间占用扩大。所以，可以在这方面进行优化。在实验最开始我尝试了理论上空间效率更高的数据结构：Bloom Filter 来减小空间占用，但发现它在频度查询的表现上较差。

(2) 查询性能：

虽然，现阶段查询功能几乎可以忽略耗时，但是这是在静态数据集上实现的。考虑到真正数据流的实时性能需求，可能需要进一步优化查询函数的实现，使用并行化技术或者索引结构等，来加速查询过程。

(3) 精度评估：

由于时间比较紧迫，我没有进行较大规模的规律性的性能测试，比如随数据量、时间戳、电影 ID、前 k 值等进行连续测试，只选取了分布平均、跨度平均的个别测试数据。以后需要对算法的查询精度进行更详细的评估，尤其在大规模数据流和高并发查询的场景下，需要有更优越的精确度和容错性。

2. 未来研究方向：

(1) 动态调整两种数据结构的参数：

由于使用数据集，所以为了获得时间和准确性的平衡，我固定了 CMsketch 和 MG 的参数。后续可以研究在不同数据流场景下，动态调整两种算法的参数，以适应数据流的变化，也能提高算法的适用性和性能。

(2) 研究增量式更新算法和实时流处理技术：

假设真的要在实时的大规模数据流上进行查询功能的实现，后续就需要研究一些更加高效的增量式更新算法，并且探索一些实时流处理技术（像 Apache Flink、Apache Kafka 等，只是初步了解，但还没有深入研究），从而实现更高效、可靠的数据流处理。

(3) 结合深度学习：

因为现阶段知识能力有限，暂未研究深度学习领域，但是人工智能离不开深度学习，所以后续我要尝试学习深度学习技术，并可以将其运用于优化这个实验中，找出更智能的数据流处理方法。

3. 总结

这是数据科学与工程算法的第一次实验课，目标是实现数据流统计算法。实验周期很长，一开始由于对课本理论知识掌握的并不充分，导致尝试了一些其他数据结构，但效果很差，后来认真钻研课本，才有了一些思路。后段实现性能评估代码时，遇到了很多问题，比如数据量很大，我怎么知道正确答案究竟是什么？测评需要得出一个准确数值或者准确规律吗？等等，现阶段代码能够使用所学知识实现任务要求的基本查询功能，并对其进行简单测试，也让我有很大收获，后续有时间，我会再深入研究其中的一些细节和优化策略。