

# Database Management Homework 4

Po-Yen Chu

## 1 Question 1

- (a) The query help to calculate the total trips per day under certain criteria, including (1) containing only trips after 2023/08/02 (2) containing only trips depart from station id  $\leq 1030$  and arrive at station 1035.
- (b) The estimated total cost is shown in Figure 1, which is 55534.57.

	QUERY PLAN text	
1	Finalize GroupAggregate (cost=55526.72..55534.57 rows=31 width=12)	
2	Group Key: travel_date	
3	-> Gather Merge (cost=55526.72..55533.95 rows=62 width=12)	
4	Workers Planned: 2	
5	-> Sort (cost=54526.69..54526.77 rows=31 width=12)	
6	Sort Key: travel_date	
7	-> Partial HashAggregate (cost=54525.62..54525.93 rows=31 width=12)	
8	Group Key: travel_date	
9	-> Parallel Seq Scan on reserved_ticket (cost=0.00..54441.72 rows=16779 width=4)	
10	Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date) AND (arrive_station_id = 10...	

Figure 1: Query Plan Before Indexing

- (c) It happened at "Parallel Seq Scan" step. This is reasonable to perform in that step since it filtered first to reduce the number of rows. There's no need to aggregating all the travel dates.
- (d) The SQL query for creating indices (respectively) are shown below:

```
-- Create index for Depart_Station_ID
CREATE INDEX idx_depart_station_id ON Reserved_Ticket (
    Depart_Station_ID);

-- Create index for Arrive_Station_ID
CREATE INDEX idx_arrive_station_id ON Reserved_Ticket (
    Arrive_Station_ID);
```

```
-- Create index for Travel_Date
CREATE INDEX idx_travel_date ON Reserved_Ticket (
    Travel_Date);
```

The query plans with indices of Depart\_Station\_ID, Arrive\_Station\_ID and Travel\_Date are shown respectively in Figure 2, 3 and 4.

	QUERY PLAN	
	text	
1	Finalize GroupAggregate (cost=55525.53..55533.39 rows=31 width=12)	
2	Group Key: travel_date	
3	-> Gather Merge (cost=55525.53..55532.77 rows=62 width=12)	
4	Workers Planned: 2	
5	-> Sort (cost=54525.51..54525.59 rows=31 width=12)	
6	Sort Key: travel_date	
7	-> Partial HashAggregate (cost=54524.43..54524.74 rows=31 width=12)	
8	Group Key: travel_date	
9	-> Parallel Seq Scan on reserved_ticket (cost=0.00..54440.54 rows=16778 width=4)	
10	Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date) AND (arrive_station_id = 10...	

Figure 2: Query Plan After Indexing on Depart\_Station\_ID

	QUERY PLAN	
	text	
1	HashAggregate (cost=35658.18..35658.49 rows=31 width=12)	
2	Group Key: travel_date	
3	-> Bitmap Heap Scan on reserved_ticket (cost=864.80..35456.85 rows=40267 width=4)	
4	Recheck Cond: (arrive_station_id = 1035)	
5	Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date))	
6	-> Bitmap Index Scan on idx_arrive_station_id (cost=0.00..854.73 rows=78174 width=4)	
7	Index Cond: (arrive_station_id = 1035)	

Figure 3: Query Plan After Indexing on Arrive\_Station\_ID

The estimated total cost comparison is shown in Table 1.

Apparently, only the index on Arrive\_Station\_ID worked and substantially reduced the total estimated cost. This is because the condition for Arrive\_Station\_ID is more specific and returned less rows, while the others returned too many rows that the DBMS chose to scan the entire table instead of scanning both the indices and the table data.


QUERY PLAN		
	text	
1	Finalize GroupAggregate (cost=55525.53..55533.39 rows=31 width=12)	
2	Group Key: travel_date	
3	-> Gather Merge (cost=55525.53..55532.77 rows=62 width=12)	
4	Workers Planned: 2	
5	-> Sort (cost=54525.51..54525.59 rows=31 width=12)	
6	Sort Key: travel_date	
7	-> Partial HashAggregate (cost=54524.43..54524.74 rows=31 width=12)	
8	Group Key: travel_date	
9	-> Parallel Seq Scan on reserved_ticket (cost=0.00..54440.54 rows=16778 width=4)	
10	Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date) AND (arrive_station_id = 10...	

Figure 4: Query Plan After Indexing on Travel\_Date

Index	Column Name	Total Estimated Cost
	Depart_Station_ID	55,533.39
	Arrive_Station_ID	35,658.49
	Travel_Date	55,533.39

Table 1: Total Estimated Costs after Indexing Respectively

- (e) According to the question, the SQL query is provided below, and the result is presented as Table 4.

```
-- 1. Let Depart_Station_ID be the first column
CREATE INDEX idx_depart_arrive ON Reserved_Ticket (
    Depart_Station_ID, Arrive_Station_ID);

-- 2. Let Arrive_Station_ID be the first column
CREATE INDEX idx_arrive_depart ON Reserved_Ticket (
    Arrive_Station_ID, Depart_Station_ID);
```

The query plans with two multi-column-indices of different orders are shown respectively in Figure 5 and 6.

Furthermore, the estimated total cost comparison is shown in Table 2.

Index	Column Name	Total Estimated Cost
	depart_arrive	55,533.39
	arrive_depart	36,487.50

Table 2: Total Estimated Costs after Multi-Indexing Respectively

	QUERY PLAN text	
1	Finalize GroupAggregate (cost=55525.53..55533.39 rows=31 width=12)	
2	Group Key: travel_date	
3	-> Gather Merge (cost=55525.53..55532.77 rows=62 width=12)	
4	Workers Planned: 2	
5	-> Sort (cost=54525.51..54525.59 rows=31 width=12)	
6	Sort Key: travel_date	
7	-> Partial HashAggregate (cost=54524.43..54524.74 rows=31 width=12)	
8	Group Key: travel_date	
9	-> Parallel Seq Scan on reserved_ticket (cost=0.00..54440.54 rows=16778 width=4)	
10	Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date) AND (arrive_station_id = 10...	

Figure 5: Query Plan After Multi-Indexing on Depart\_Arrive

	QUERY PLAN text	
1	HashAggregate (cost=36487.19..36487.50 rows=31 width=12)	
2	Group Key: travel_date	
3	-> Bitmap Heap Scan on reserved_ticket (cost=571.14..36285.85 rows=40267 width=...	
4	Recheck Cond: ((arrive_station_id = 1035) AND (depart_station_id <= 1030))	
5	Filter: (travel_date >= '2023-08-02'::date)	
6	-> Bitmap Index Scan on idx_arrive_depart (cost=0.00..561.07 rows=41664 width=...	
7	Index Cond: ((arrive_station_id = 1035) AND (depart_station_id <= 1030))	

Figure 6: Query Plan After Multi-Indexing on Arrive\_Depart

The order of multi-indexing matters significantly. In this case, prioritizing the filter on `arrive_station_id` is more effective because the data range it targets is smaller compared to the other index. This reasoning aligns closely with the analysis provided in the previous question's results.

- (f) The real execution times (recorded in the first execution) and estimated total costs for the query with different or no indices are shown in Table 3. The real execution times are not proportional to estimated total cost. In this table, a conclusion may be drawn that `arrive_depart` has a better performance. However, execution times varies a lot even with the same index. The data size in this query is inadequate to observe stable differences.

The interpretation to the variety of real execution times is that the estimated cost is a theoretical value determined during the planning phase, based on the estimated cost of page reads and computations. However, the actual execution time is influenced by other system-level factors, including disk I/O, memory usage,

Index Column Name	Total Estimated Cost	Execution Time
No Index	55,534.57	420msec
Depart_Station_ID	55,533.39	415msec
Arrive_Station_ID	35,658.49	482msec
Travel_Date	55,533.39	434msec
depart_arrive	55,533.39	454msec
arrive_depart	36,487.50	385msec

Table 3: Total Estimated Costs and Execution Times

and data distribution.

Furthermore, it appears that using `arrive_station_id` as the first column in the index is the only way to improve theoretical efficiency, regardless of whether a single-column or multi-column index is used. However, multi-column indexes seem to perform worse than their single-column counterparts in this scenario. This could be due to the inclusion of `depart_station_id` in the multi-column index, which adds unnecessary complexity and does not significantly contribute to filtering efficiency. As a result, the additional overhead may outweigh the potential benefits, leading to suboptimal performance.

## 2 Question 2

- (a) The B+ tree is shown in Figure 7.

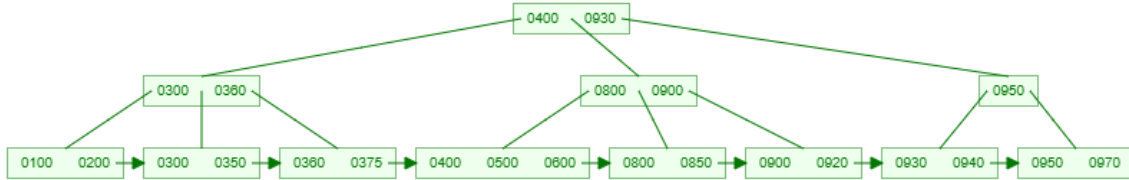


Figure 7: Complete B+ Tree by Top-down Insertion

- (b) The tree before and after the insertion of 360 is shown respectively in Figure 8 and 9. 360 is at first inserted to the leaf node between 350 and 375, which exceeded the limit, 360 is copied up to the parent node between 300 and 400 since it is the middle value, Now the parent node has four values exceeding the limit, so the middle value (400) is pushed up to become a new root node.
- (c) The tree before and after the insertion of 930 is shown respectively in Figure 11 and 7. 930 is at first inserted to the leaf node between 920 and 940, which exceeded the limit, 930 is copied up to the parent node between 900 and 950 since it is the middle value, it. Now the parent node has four values exceeding the limit, so the middle value (930) is pushed up to the root node next to 400.
- (d) The bottom-up constructed B+ tree is shown in Figure ??

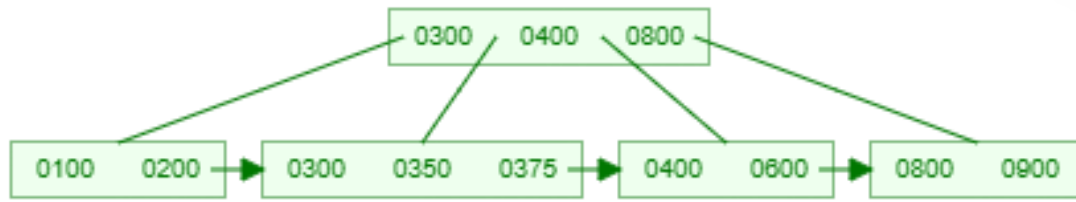


Figure 8: B+ Tree Before the Insertion of 360

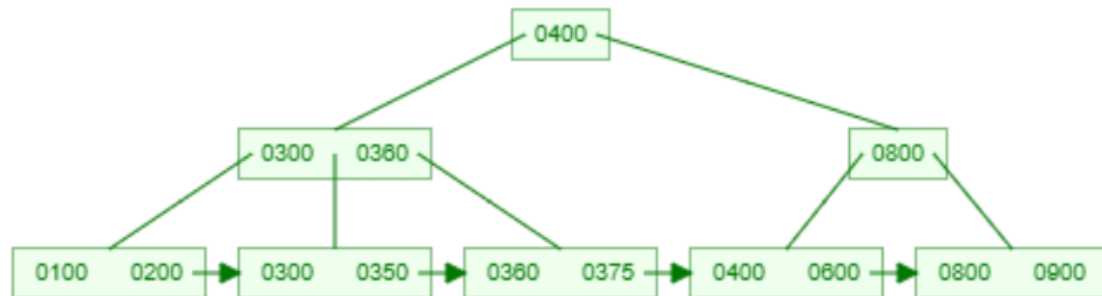


Figure 9: B+ Tree After the Insertion of 360

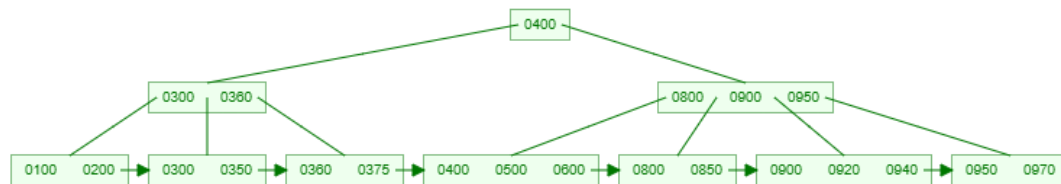


Figure 10: B+ Tree Before the Insertion of 930

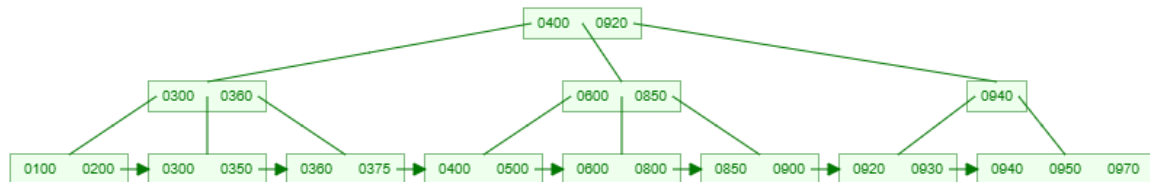


Figure 11: Complete B+ Tree by Bottom-up Construction

- (e) The two trees (top-down and bottom-up) will be identical, since the assumption is that the numbers are already sorted. The sorted elements ensure that the tree remains balanced after each insertion, and regardless of whether the insertion starts from the top-down or bottom-up, the result will be the same.

### 3 Question 3

The SQL Query for finding total count of customers' reserved and non-reserved tickets is shown below:

**SELECT**

```

        m.citizen_id,
        COALESCE(COUNT(rt.ticket_id), 0) AS reserved_ticket_count,
        COALESCE(COUNT(nrt.ticket_id), 0) AS
            non_reserved_ticket_count
FROM
    member m
LEFT JOIN reserved_ticket rt ON m.citizen_id = rt.citizen_id
LEFT JOIN non_reserved_ticket nrt ON m.citizen_id = nrt.
    citizen_id
GROUP BY
    m.citizen_id
ORDER BY
    m.citizen_id;

```

Then create indices on citizen\_id of three tables:

```

CREATE INDEX idx_citizen_id ON member(citizen_id);
CREATE INDEX idx_citizen_r_id ON reserved_ticket(citizen_id);
CREATE INDEX idx_citizen_n_id ON non_reserved_ticket(
    citizen_id);

```

The query plans before and after creating indices are shown respectively in Figure 12 and 13.

The real execution times (recorded in the first execution) and estimated total costs for the query with different or no indices are shown in Table 4.

Index Column Name	Total Estimated Cost	Execution Time
Before Indexing	359,070.72	25.455sec
After Indexing	470,352.09	18.253msec

Table 4: Total Estimated Costs and Execution Times

After indexing, the join method switched to a nested loop join from a parallel hash right join because the indices allowed the database to efficiently fetch matching rows directly via index scans, reducing the need for hashing and parallel full table scans.

## 4 Question 4

- (a) • Stupid Nested Loop Join (SNLJ):

$$\text{Cost} = M + mN = 5,000 + 500,000 \times 40,000 = 20,000,005,000 \text{ I/O.}$$

$$\text{Time} = \text{Cost} \times 0.1 \text{ ms} = 2,000,000.5 \text{ seconds.}$$

	QUERY PLAN text	
1	Finalize GroupAggregate (cost=310473.00..470352.09 rows=300000 width=26)	
2	Group Key: m.citizen_id	
3	-> Gather Merge (cost=310473.00..462852.09 rows=600000 width=26)	
4	Workers Planned: 2	
5	-> Partial GroupAggregate (cost=309472.98..392597.18 rows=300000 width=26)	
6	Group Key: m.citizen_id	
7	-> Merge Left Join (cost=309472.98..364736.06 rows=3314815 width=18)	
8	Merge Cond: ((m.citizen_id)::text = (nrt.citizen_id)::text)	
9	-> Sort (cost=171323.35..173819.44 rows=998435 width=14)	
10	Sort Key: m.citizen_id	
11	-> Parallel Hash Right Join (cost=6236.50..54766.75 rows=998435 width=14)	
12	Hash Cond: ((rt.citizen_id)::text = (m.citizen_id)::text)	
13	-> Parallel Seq Scan on reserved_ticket rt (cost=0.00..45347.74 rows=1212374 width=...	
14	-> Parallel Hash (cost=4674.00..4674.00 rows=125000 width=10)	
15	-> Parallel Seq Scan on member m (cost=0.00..4674.00 rows=125000 width=10)	
16	-> Materialize (cost=138148.34..143148.34 rows=1000000 width=14)	
17	-> Sort (cost=138148.34..140648.34 rows=1000000 width=14)	
18	Sort Key: nrt.citizen_id	
19	-> Seq Scan on non_reserved_ticket nrt (cost=0.00..21400.00 rows=1000000 width=14)	

Figure 12: Query Plan Before Indexing

- **Single-Block Nested Loop Join (SBNLJ):**

$$\text{Cost} = M + MN = 5,000 + 5,000 \times 40,000 = 200,005,000 \text{ I/O.}$$

$$\text{Time} = \text{Cost} \times 0.1 \text{ ms} = 20,000.5 \text{ seconds.}$$

- **Multi-Block Nested Loop Join (MBNLJ):**

$$\text{Cost} = M + \left\lceil \frac{M}{B-2} \right\rceil \times N = 5,000 + \left\lceil \frac{5,000}{98} \right\rceil \times 40,000.$$

$$\text{Cost} = 5,000 + 52 \times 40,000 = 2,085,000 \text{ I/O.}$$

$$\text{Time} = \text{Cost} \times 0.1 \text{ ms} = 208.5 \text{ seconds.}$$

- (b)
- **Stupid Nested Loop Join:** High cost due to repeated scanning of both  $R$  and  $S$ , accessing  $S$   $m$  times.
  - **Single-Block Nested Loop Join:** Reduces redundant scans of  $R$ , but still accesses  $S$   $M$  times.
  - **Multi-Block Nested Loop Join:** Minimizes  $R$ 's page reads by processing multiple pages in memory, leveraging buffer space effectively.



QUERY PLAN		text	
1			
1	Finalize GroupAggregate	(cost=1001.30..359070.72 rows=300000 width=26)	
2	Group Key: m.citizen_id		
3	-> Gather Merge	(cost=1001.30..351570.72 rows=600000 width=26)	
4	Workers Planned: 2		
5	-> Partial GroupAggregate	(cost=1.28..281315.81 rows=300000 width=26)	
6	Group Key: m.citizen_id		
7	-> Merge Left Join	(cost=1.28..253454.70 rows=3314815 width=18)	
8	Merge Cond: ((m.citizen_id)::text = (nrt.citizen_id)::text)		
9	-> Nested Loop Left Join	(cost=0.85..150847.42 rows=998435 width=14)	
10	-> Parallel Index Only Scan using idx_citizen_id on member m	(cost=0.42..7407.42 rows=125000 width=10)	
11	-> Index Scan using idx_citizen_r_id on reserved_ticket rt	(cost=0.43..1.07 rows=8 width=14)	
12	Index Cond: ((citizen_id)::text = (m.citizen_id)::text)		
13	-> Materialize	(cost=0.42..72349.84 rows=1000000 width=14)	
14	-> Index Scan using idx_citizen_n_id on non_reserved_ticket nrt	(cost=0.42..69849.84 rows=1000000 width=14)	

Figure 13: Query Plan After Indexing

## 5 Question 5

- (a) The query will return five columns: employee's first name and last name, employee's department number and its corresponding department name and location. It is implemented by joining **DEPARTMENT** and **DEPT\_LOACATION** on **EMPLOYEE**. The first five rows of the result is shown in Table 5.

fname	lname	dno	dname	dlocation
James	Borg	1	Headquarters	Houston
Franklin	Wong	5	Research	Bellaire
Franklin	Wong	5	Research	Sugarland
Franklin	Wong	5	Research	Houston
Jennifer	Wallace	4	Administration	Stafford

Table 5: First 5 Rows of the Query's Result

- (b) The three types of join method is shown in Figure 14.
- (c) The query plan (Figure 15) corresponds to the tree on the left hand side of Figure 14, which **department** and **dept\_locations** are joined first. The planner likely chose this join tree because **department** has only 3 rows, making it efficient to scan and hash. **dept\_locations** has 5 rows, resulting in a small intermediate result (at most 5 rows) from the hash join. Joining employee (8 rows) with the smaller intermediate result minimizes the computational cost.
- (d) The query plan is identical to Figure 15, The query plan is the same because the queries represent logically equivalent operations, and the query planner optimizes

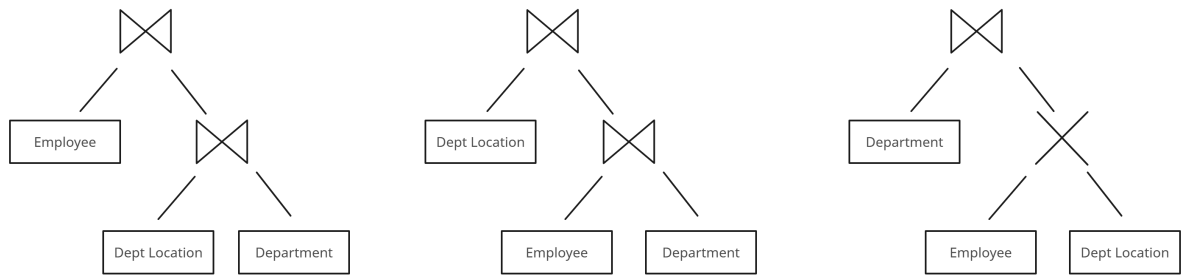


Figure 14: Three Types of Join Method


	QUERY PLAN	
	text	
1	Hash Join (cost=2.22..3.47 rows=13 width=36)	
2	Hash Cond: (e.dno = d.dnumber)	
3	-> Seq Scan on employee e (cost=0.00..1.08 rows=8 width=16)	
4	-> Hash (cost=2.15..2.15 rows=5 width=28)	
5	-> Hash Join (cost=1.07..2.15 rows=5 width=28)	
6	Hash Cond: (dl.dnumber = d.dnumber)	
7	-> Seq Scan on dept_locations dl (cost=0.00..1.05 rows=5 width=...	
8	-> Hash (cost=1.03..1.03 rows=3 width=16)	
9	-> Seq Scan on department d (cost=0.00..1.03 rows=3 width=...	

Figure 15: Query Plan After Analyzing

both into the same execution strategy. The subquery in the second query does not introduce additional complexity or constraints, so the planner treats it as a single join tree.

## 6 Question 6

(a) Total records in buckets:

- Bucket 60: 10
- Bucket 70: 1
- Bucket 80: 0

The implementation is shown below:

```
import csv
```

```

# Step 1: read csv
def read_csv(file_path):
    with open(file_path, mode='r', encoding='utf-8') as
        file:
            reader = csv.DictReader(file)
            data = [row for row in reader]
    return data

reserved_ticket_data = read_csv('arrive_citizen.csv')
member_data = read_csv('citizen_info.csv')

# Step 2: define hash function
def hash_function(citizen_id):
    return sum(ord(char) for char in citizen_id) % 100

# Step 3: initialize 100 bucket
buckets = {i: [] for i in range(100)}

# reserved_ticket buckets
for ticket in reserved_ticket_data:
    bucket_index = hash_function(ticket["citizen_id"])
    buckets[bucket_index].append({
        "arrive_station_id": ticket["arrive_station_id"],
        "citizen_id": ticket["citizen_id"],
        "name": None # temp value
    })

# Step 4: member hash join
for member in member_data:
    bucket_index = hash_function(member["citizen_id"])
    for ticket in buckets[bucket_index]:
        if ticket["citizen_id"] == member["citizen_id"]:
            ticket["name"] = member["name"]

# Step 5: print all buckets
def print_results():
    for bucket_index in range(100):
        count = 0
        print(f"\nBucket {bucket_index}:")
        for ticket in buckets[bucket_index]:
            if ticket["name"]: # print only name not None
                count += 1
                print(f"arrive_station_id: {ticket['arrive_station_id']}, name: {ticket['name']}")
        print(f"Total records in Bucket {bucket_index}: {count}")

```

```

        count}"))

print_results()

```

(b) The implementation is shown below:

```

import csv

# read csv files
def read_csv(file_path):
    with open(file_path, mode='r', encoding='utf-8') as file:
        reader = csv.DictReader(file)
        data = [row for row in reader]
    return data

reserved_ticket_data = read_csv('arrive_citizen.csv') #
    21,290 rows
member_data = read_csv('citizen_info.csv') # 10,248 rows

# stupid join
for member in member_data:
    for ticket in reserved_ticket_data:
        ticket["name"] = member["name"] # fill name

# print name not null
count = 0
for ticket in reserved_ticket_data:
    if ticket.get("name"): # check if name is not None
        count += 1
        print(f"arrive_station_id: {ticket['
            arrive_station_id']}, name: {ticket['name']}")

print(f"Total records with name: {count}")

```

(c) The execution time output is shown below:

```

Stupid Join Execution Time: 39.38227653503418 sec
Hash Join Execution Time: 2.440889596939087 sec

```

For the Stupid Join, the complexity is  $O(n * m)$  because it performs a nested loop, comparing each record in one dataset with every record in the other. In contrast, Hash Join has a complexity of  $O(n + m)$  because it first builds a hash table for one dataset ( $O(n)$ ) and then performs a single pass to find matches using the hash table ( $O(m)$ ). This makes Hash Join more efficient, as it avoids the need for a costly nested loop.

The disadvantages of Hash Join include memory usage and hash collisions. Hash Join requires additional memory to store the hash table (buckets), and with large datasets, this can lead to memory shortages. Additionally, hash collisions can impact performance; although modern hashing algorithms minimize collisions, when buckets contain many records, the search time can increase due to the need to handle collisions.