Database Management
Homework 4

B11705044
Yen-Hung, Chiang

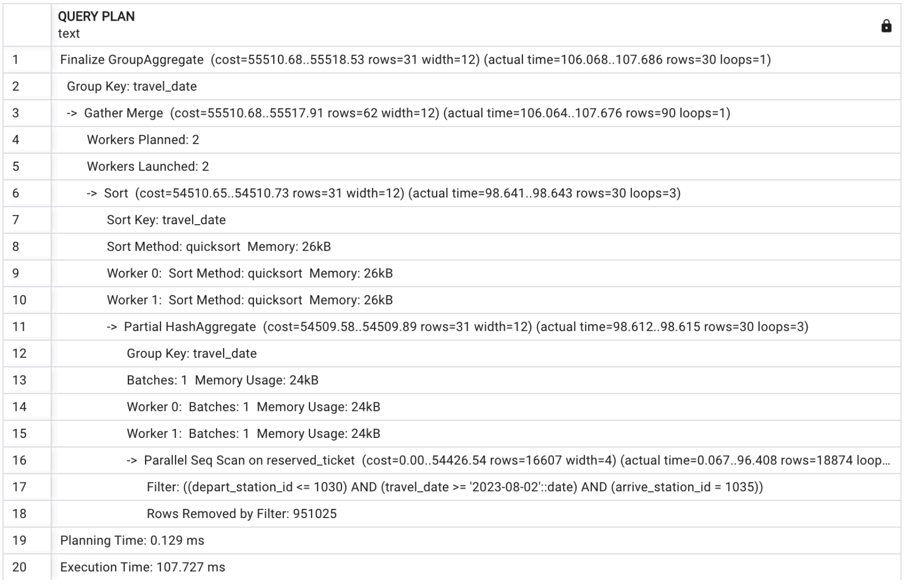
1

(a)

這句指令從 `Reserved_Ticket` 這個表中查詢日期大於等於 2023-08-02 的 `Travel_Date` 以及計算每個 `Travel_Date` 對應的票數，查詢條件是 `Depart_Station_ID` $\leq$ 1030 及 `Arrive_Station_ID` $=$ 1035。

(b)

圖 1 是在沒有任何 index 的情況下執行的 query plan，可以看到 RDBMS 產出的 estimated total cost 為 55518.53。



```
    QUERY PLAN
    text
1   Finalize GroupAggregate  (cost=55510.68..55518.53 rows=31 width=12) (actual time=106.068..107.686 rows=30 loops=1)
2     Group Key: travel_date
3     ->  Gather Merge  (cost=55510.68..55517.91 rows=62 width=12) (actual time=106.064..107.676 rows=90 loops=1)
4           Workers Planned: 2
5           Workers Launched: 2
6           ->  Sort  (cost=54510.65..54510.73 rows=31 width=12) (actual time=98.641..98.643 rows=30 loops=3)
7                 Sort Key: travel_date
8                 Sort Method: quicksort  Memory: 26kB
9                 Worker 0:  Sort Method: quicksort  Memory: 26kB
10                Worker 1:  Sort Method: quicksort  Memory: 26kB
11                ->  Partial HashAggregate  (cost=54509.58..54509.89 rows=31 width=12) (actual time=98.612..98.615 rows=30 loops=3)
12                      Group Key: travel_date
13                      Batches: 1  Memory Usage: 24kB
14                      Worker 0:  Batches: 1  Memory Usage: 24kB
15                      Worker 1:  Batches: 1  Memory Usage: 24kB
16                      ->  Parallel Seq Scan on reserved_ticket  (cost=0.00..54426.54 rows=16607 width=4) (actual time=0.067..96.408 rows=18874 loop...
17                            Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date) AND (arrive_station_id = 1035))
18                            Rows Removed by Filter: 951025
19  Planning Time: 0.129 ms
20  Execution Time: 107.727 ms
```

圖 1: Query plan

(c)

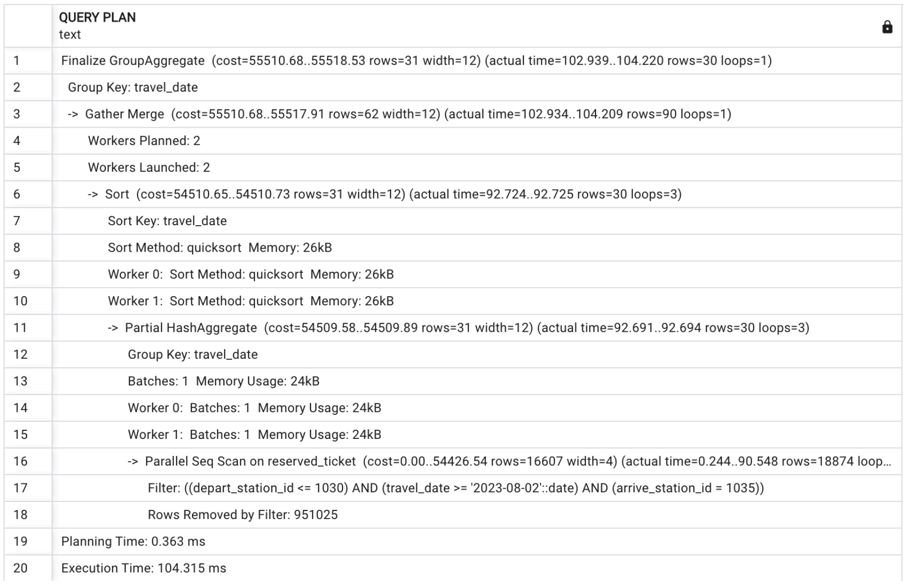
篩選 `Travel_Date` $\geq$ '2023-08-02' 在第 10 行發生，代表 RDBMS 在最一開始就先篩選條件。RDBMS 在最一開始就篩選可以讓他之後再做其他運算的時候處理較少的資料，所以在這個階段篩選

合理。

(d)

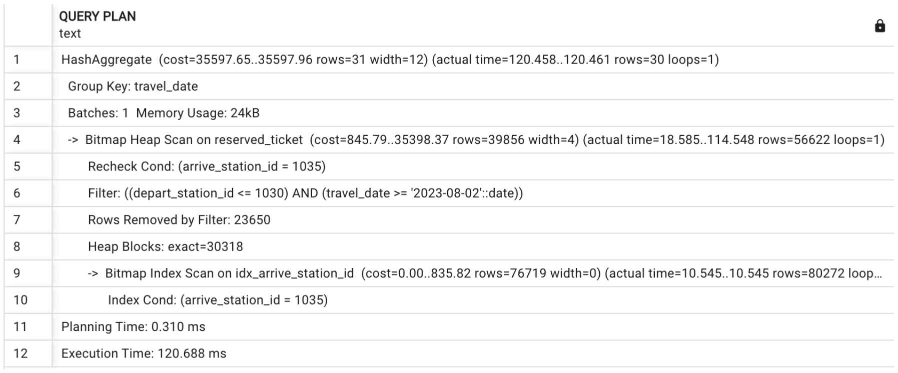
以下的 sql 程式碼為建立三個 index 的語法。以下分別幫 `Depart_Station_ID` 、`Arrive_Station_ID`、`Travel_Date` 建立 index，圖 2、圖 3、圖 4 分別為在只建立該 index 的情況所產出的 query plan。

```sql
CREATE INDEX IF NOT EXISTS idx_depart_station_id ON Reserved_Ticket(Depart_Station_ID);
CREATE INDEX IF NOT EXISTS idx_arrive_station_id ON Reserved_Ticket(Arrive_Station_ID);
CREATE INDEX IF NOT EXISTS idx_travel_date ON Reserved_Ticket(Travel_Date);
```



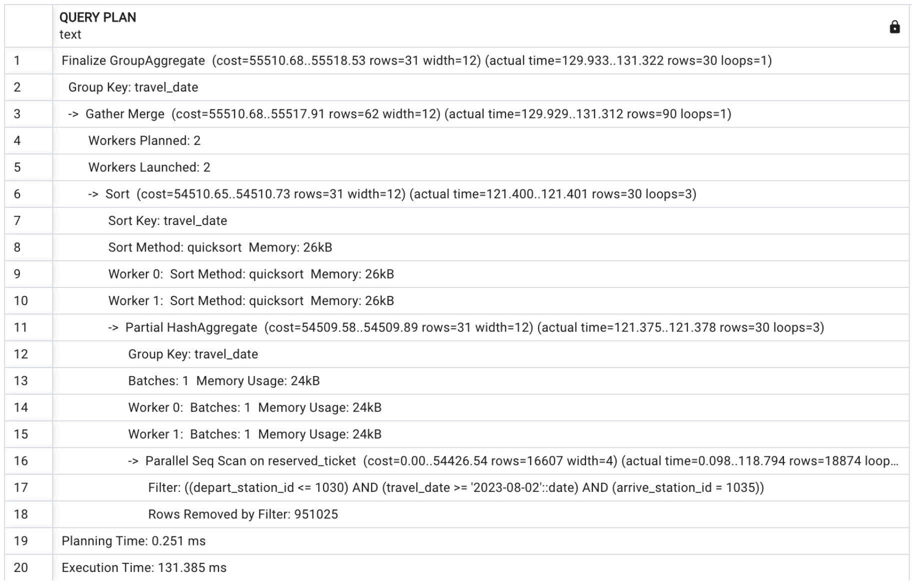
```
QUERY PLAN
text
Finalize GroupAggregate  (cost=55510.68..55518.53 rows=31 width=12) (actual time=102.939..104.220 rows=30 loops=1)
  Group Key: travel_date
  ->  Gather Merge  (cost=55510.68..55517.91 rows=62 width=12) (actual time=102.934..104.209 rows=90 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Sort  (cost=54510.65..54510.73 rows=31 width=12) (actual time=92.724..92.725 rows=30 loops=3)
              Sort Key: travel_date
              Sort Method: quicksort  Memory: 26kB
              Worker 0:  Sort Method: quicksort  Memory: 26kB
              Worker 1:  Sort Method: quicksort  Memory: 26kB
              ->  Partial HashAggregate  (cost=54509.58..54509.89 rows=31 width=12) (actual time=92.691..92.694 rows=30 loops=3)
                    Group Key: travel_date
                    Batches: 1  Memory Usage: 24kB
                    Worker 0:  Batches: 1  Memory Usage: 24kB
                    Worker 1:  Batches: 1  Memory Usage: 24kB
                    ->  Parallel Seq Scan on reserved_ticket  (cost=0.00..54426.54 rows=16607 width=4) (actual time=0.244..90.548 rows=18874 loop...
                          Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date) AND (arrive_station_id = 1035))
                          Rows Removed by Filter: 951025
Planning Time: 0.363 ms
Execution Time: 104.315 ms
```

圖 2: Query plan for index on `Depart_Station_ID`



```
QUERY PLAN
text
HashAggregate  (cost=35597.65..35597.96 rows=31 width=12) (actual time=120.458..120.461 rows=30 loops=1)
  Group Key: travel_date
  Batches: 1  Memory Usage: 24kB
  ->  Bitmap Heap Scan on reserved_ticket  (cost=845.79..35398.37 rows=39856 width=4) (actual time=18.585..114.548 rows=56622 loops=1)
        Recheck Cond: (arrive_station_id = 1035)
        Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date))
        Rows Removed by Filter: 23650
        Heap Blocks: exact=30318
        ->  Bitmap Index Scan on idx_arrive_station_id  (cost=0.00..835.82 rows=76719 width=0) (actual time=10.545..10.545 rows=80272 loop...
              Index Cond: (arrive_station_id = 1035)
Planning Time: 0.310 ms
Execution Time: 120.688 ms
```

圖 3: Query plan for index on `Arrive_Station_ID`



```
QUERY PLAN
text
1  Finalize GroupAggregate  (cost=55510.68..55518.53 rows=31 width=12) (actual time=129.933..131.322 rows=30 loops=1)
2    Group Key: travel_date
3    ->  Gather Merge  (cost=55510.68..55517.91 rows=62 width=12) (actual time=129.929..131.312 rows=90 loops=1)
4          Workers Planned: 2
5          Workers Launched: 2
6          ->  Sort  (cost=54510.65..54510.73 rows=31 width=12) (actual time=121.400..121.401 rows=30 loops=3)
7                Sort Key: travel_date
8                Sort Method: quicksort  Memory: 26kB
9                Worker 0:  Sort Method: quicksort  Memory: 26kB
10               Worker 1:  Sort Method: quicksort  Memory: 26kB
11               ->  Partial HashAggregate  (cost=54509.58..54509.89 rows=31 width=12) (actual time=121.375..121.378 rows=30 loops=3)
12                     Group Key: travel_date
13                     Batches: 1  Memory Usage: 24kB
14                     Worker 0:  Batches: 1  Memory Usage: 24kB
15                     Worker 1:  Batches: 1  Memory Usage: 24kB
16                     ->  Parallel Seq Scan on reserved_ticket  (cost=0.00..54426.54 rows=16607 width=4) (actual time=0.098..118.794 rows=18874 loop...
17                           Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date) AND (arrive_station_id = 1035))
18                           Rows Removed by Filter: 951025
19 Planning Time: 0.251 ms
20 Execution Time: 131.385 ms
```

圖 4: Query plan for index on `Travel_Date`

Index On	Total Cost
`Depart_Station_ID`	55518.53
`Arrive_Station_ID`	35597.96
`Travel_Date`	55518.53

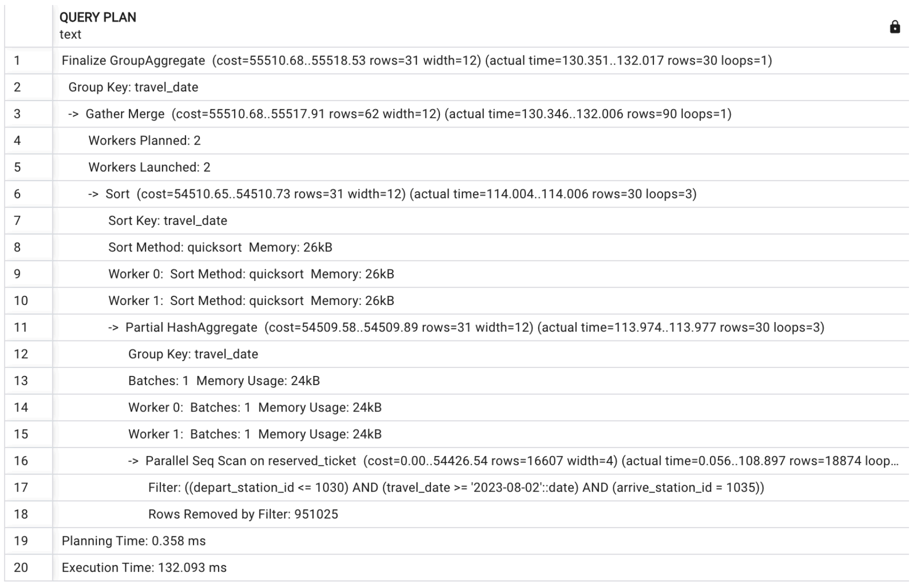
表 1: Comparison of Total Cost on Different Indexes

表 2 統整了使用不同的 index 所得到的 estimated total cost 差異，我們可以看到使用 `Arrive_Station_ID` 建立 index 所產出的 estimated total cost 是最小的，另外兩者則是差不多。回到上面的三張圖可以看到只有建立 `Arrive_Station_ID` 當 index 的時候 RDBMS 會使用 index，其他兩者不會，猜測是因爲另外兩者建立的 index 查詢速度沒有比原來的快，所以 RDBMS 沒有採用 index。

(e)

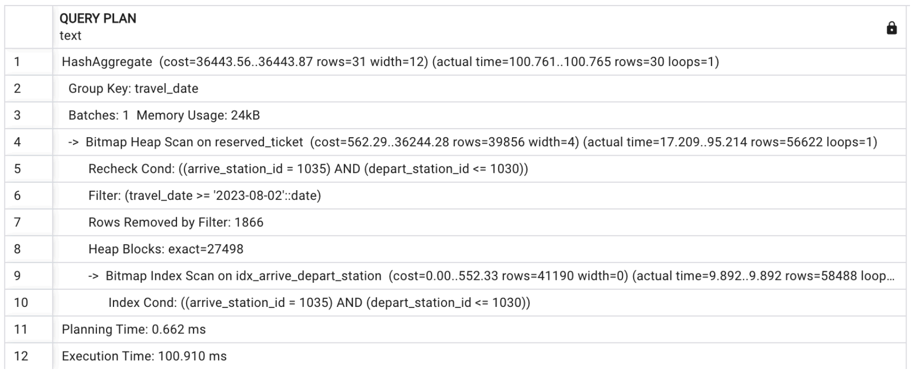
以下的 sql 程式碼為建立兩個 multi-column index 的語句，圖 5 是對 (`Depart_Station_ID`, `Arrive_Station_ID`) 建立 index 之後產生的 query plan，圖 6 是對 (`Arrive_Station_ID`, `Depart_Station_ID`) 建立 index 產生的 query plan。

```sql
CREATE INDEX IF NOT EXISTS idx_depart_arrive_station ON Reserved_Ticket
↪  (Depart_Station_ID, Arrive_Station_ID);
CREATE INDEX IF NOT EXISTS idx_arrive_depart_station ON Reserved_Ticket
↪  (Arrive_Station_ID, Depart_Station_ID);
```



```
QUERY PLAN
text
Finalize GroupAggregate  (cost=55510.68..55518.53 rows=31 width=12) (actual time=130.351..132.017 rows=30 loops=1)
  Group Key: travel_date
  ->  Gather Merge  (cost=55510.68..55517.91 rows=62 width=12) (actual time=130.346..132.006 rows=90 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Sort  (cost=54510.65..54510.73 rows=31 width=12) (actual time=114.004..114.006 rows=30 loops=3)
              Sort Key: travel_date
              Sort Method: quicksort  Memory: 26kB
              Worker 0:  Sort Method: quicksort  Memory: 26kB
              Worker 1:  Sort Method: quicksort  Memory: 26kB
              ->  Partial HashAggregate  (cost=54509.58..54509.89 rows=31 width=12) (actual time=113.974..113.977 rows=30 loops=3)
                    Group Key: travel_date
                    Batches: 1  Memory Usage: 24kB
                    Worker 0:  Batches: 1  Memory Usage: 24kB
                    Worker 1:  Batches: 1  Memory Usage: 24kB
                    ->  Parallel Seq Scan on reserved_ticket  (cost=0.00..54426.54 rows=16607 width=4) (actual time=0.056..108.897 rows=18874 loop…
                          Filter: ((depart_station_id <= 1030) AND (travel_date >= '2023-08-02'::date) AND (arrive_station_id = 1035))
                          Rows Removed by Filter: 951025
Planning Time: 0.358 ms
Execution Time: 132.093 ms
```

圖 5: Query plan for index on (`Depart_Station_ID`, `Arrive_Station_ID`)



```
QUERY PLAN
text
HashAggregate  (cost=36443.56..36443.87 rows=31 width=12) (actual time=100.761..100.765 rows=30 loops=1)
  Group Key: travel_date
  Batches: 1  Memory Usage: 24kB
  ->  Bitmap Heap Scan on reserved_ticket  (cost=562.29..36244.28 rows=39856 width=4) (actual time=17.209..95.214 rows=56622 loops=1)
        Recheck Cond: ((arrive_station_id = 1035) AND (depart_station_id <= 1030))
        Filter: (travel_date >= '2023-08-02'::date)
        Rows Removed by Filter: 1866
        Heap Blocks: exact=27498
        ->  Bitmap Index Scan on idx_arrive_depart_station  (cost=0.00..552.33 rows=41190 width=0) (actual time=9.892..9.892 rows=58488 loop…
              Index Cond: ((arrive_station_id = 1035) AND (depart_station_id <= 1030))
Planning Time: 0.662 ms
Execution Time: 100.910 ms
```

圖 6: Query plan for index on (`Arrive_Station_ID`, `Depart_Station_ID`)

Index On	**Total Cost**
(`Depart_Station_ID`, `Arrive_Station_ID`)	55518.53
(`Arrive_Station_ID`, `Depart_Station_ID`)	36443.87

表 2: Comparison of Total Cost on Two Different Indexes

根據圖 5 和圖 6 的 query plan，可以把 estimated total cost 整理成表 2，可以明顯地看到 `Arrive_Station_ID` 在前面的 cost 比較小，而回去看 query plan 也會發現 `Arrive_Station_ID` 在前面的 index 因為 cost 比較小所以會被 RDBMS 使用，而另外一個 index 則不會被使用。

(f)

Index On	Total Cost	Execution Time
None	55518.53	146ms
`Depart_Station_ID`	55518.53	104ms
`Arrive_Station_ID`	35597.96	120ms
`Travel_Date`	55518.53	131ms
(`Depart_Station_ID`, `Arrive_Station_ID`)	55518.53	132ms
(`Arrive_Station_ID`, `Depart_Station_ID`)	36443.87	100ms

表 3: Comparison of Different Indexes

比較六種執行該 SQL 指令的方法並且把結果整理成表 3，觀察到除了 `Arrive_Station_ID` 和 (`Arrive_Station_ID`, `Depart_Station_ID`) 這兩種建立 index 的方式，其他的 cost 都和沒有建立 index 時一樣大，推測是因為這種 index 方式並不會幫助增進查詢的效率。另外觀察到 estimated total cost 比較小的 index 方式在實際執行時並沒有比起其他方式有明顯的差異，有可能是因為資料量不夠大讓執行時間差異不容易觀察到。

2

(a)

圖 7 是把數列的值由左至右依序插入畫出的 B+ tree。

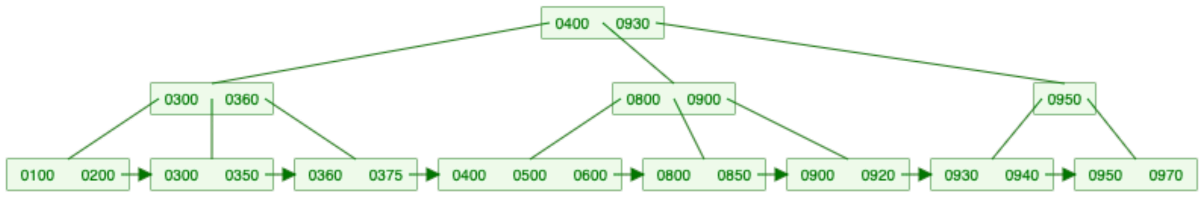


圖 7: B+ tree

(b)

圖 8 以及圖 9 分別是插入 360 前以及插入 360 後的 B+ tree。可以觀察到在插入 360 之後由於左邊第二個 leaf node 裡面會有四個值，所以把 360 copy up 到上面的 inner node，接著 inner node 也會有四個值，所以再把 400 push up 到更上面的 inner node，最後就形成如圖 9 的 B+ tree。

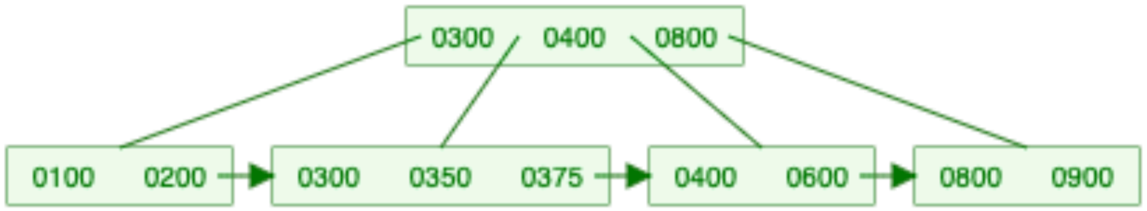


圖 8: B+ tree before insert 360

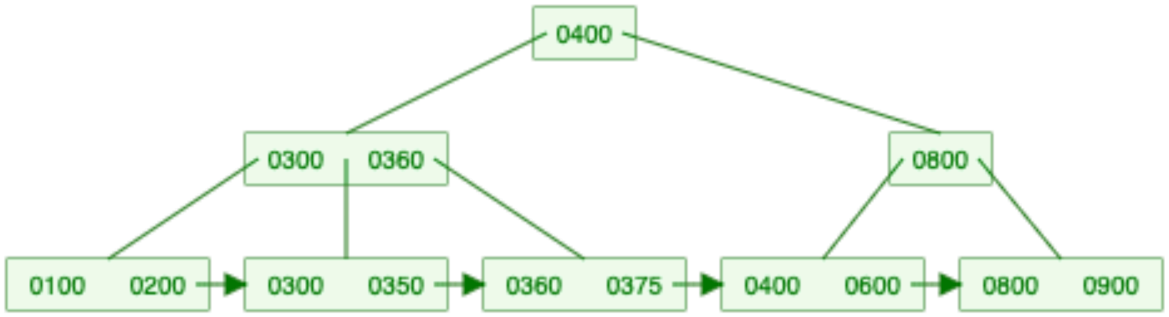


圖 9: B+ tree after insert 360

(c)

圖 10 以及圖 11 分別是插入 930 前以及插入 930 後的 B+ tree。在插入 930 的時候會先插入到 900 所在的 leaf node，接著裡面超過 key 所以把 930 copy up 到上面的 inner node，再來 inner node 裡面的 key 數量也會有四個，所以把 930 push up 到最上面的 inner node，就形成了如圖 11 的 B+ tree。

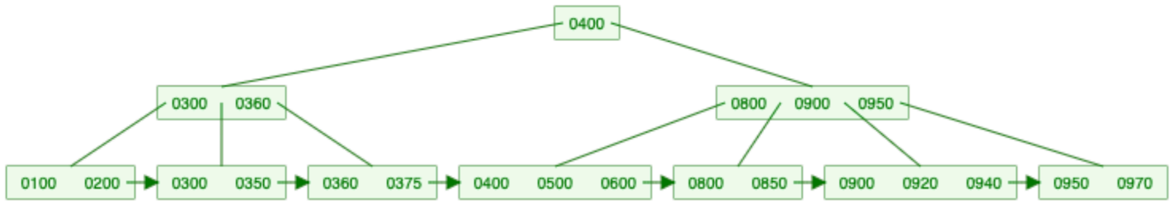


圖 10: B+ tree before insert 930

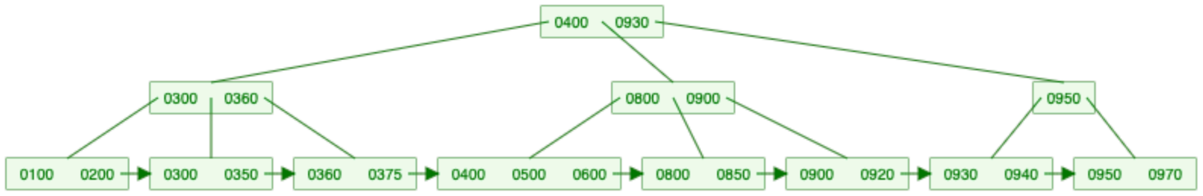


圖 11: B+ tree after insert 930

(d)

把所有值排序之後用 bottom up 方式建構的 B+ tree 如圖 12 所示。

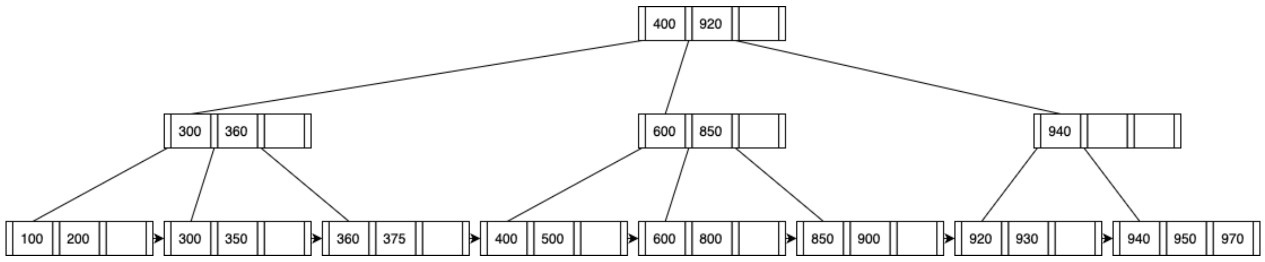


圖 12: B+ tree bottom up

(e)

使用兩種方式建構的 B+ tree 應該會長得一樣，因為用 top-down 的方式建立的 tree（圖 13）在 leaf node 裡面的值變成四個的時候會分裂成兩個兩個並且 copy up 中間的值上去 inner node，而在 inner node 裡面的值達到四個的時候會把中間的值 push up 上去，所以最後會形成一樣的圖形。

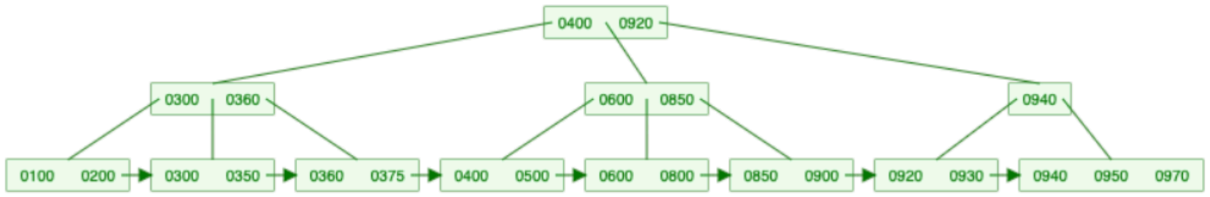


圖 13: B+ tree top-down (sorted)

3

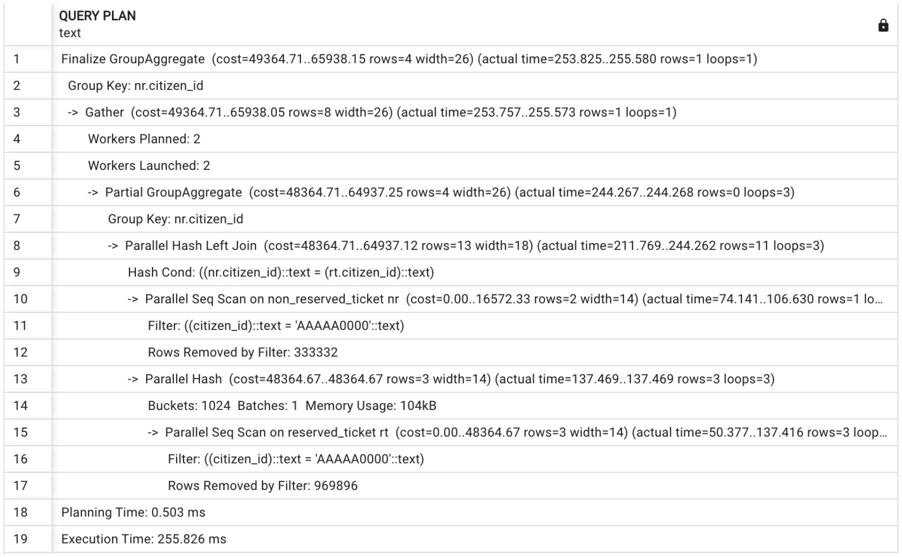
我想要查詢 `citizen_id` 是 `'AAAAA0000'` 購買的 `reserved_ticket` 和 `non_reserved_ticket` 的數量，以下是執行這個查詢的 SQL 程式碼。這個程式碼會 join `reserved_ticket` 和 `non_reserved_ticket` 這兩張表並且計算這兩個表的 ticket 數量。

```
SELECT
    nr.citizen_id,
```

```sql
    COUNT(nr.ticket_id) AS non_reserved_ticket_count,
    COUNT(rt.ticket_id) AS reserved_ticket_count
FROM non_reserved_ticket nr
LEFT JOIN reserved_ticket rt ON nr.citizen_id = rt.citizen_id
GROUP BY nr.citizen_id
HAVING nr.citizen_id = 'AAAAA0000';
```

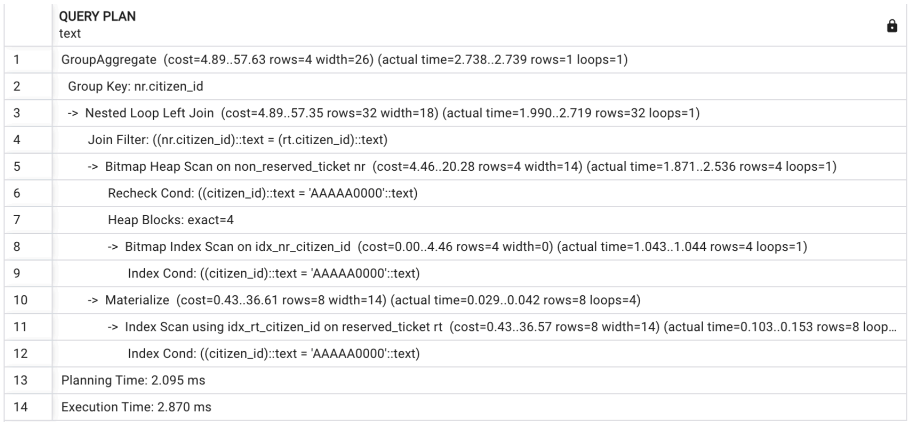
下面兩句 SQL 程式碼是建立 index 的語句，分別對兩個資料表的 `citizen_id` 建構 index。

```sql
CREATE INDEX idx_nr_citizen_id ON non_reserved_ticket (citizen_id);
CREATE INDEX idx_rt_citizen_id ON reserved_ticket (citizen_id);
```



	QUERY PLAN text
1	Finalize GroupAggregate (cost=49364.71..65938.15 rows=4 width=26) (actual time=253.825..255.580 rows=1 loops=1)
2	Group Key: nr.citizen_id
3	-> Gather (cost=49364.71..65938.05 rows=8 width=26) (actual time=253.757..255.573 rows=1 loops=1)
4	Workers Planned: 2
5	Workers Launched: 2
6	-> Partial GroupAggregate (cost=48364.71..64937.25 rows=4 width=26) (actual time=244.267..244.268 rows=0 loops=3)
7	Group Key: nr.citizen_id
8	-> Parallel Hash Left Join (cost=48364.71..64937.12 rows=13 width=18) (actual time=211.769..244.262 rows=11 loops=3)
9	Hash Cond: ((nr.citizen_id)::text = (rt.citizen_id)::text)
10	-> Parallel Seq Scan on non_reserved_ticket nr (cost=0.00..16572.33 rows=2 width=14) (actual time=74.141..106.630 rows=1 lo...
11	Filter: ((citizen_id)::text = 'AAAAA0000'::text)
12	Rows Removed by Filter: 333332
13	-> Parallel Hash (cost=48364.67..48364.67 rows=3 width=14) (actual time=137.469..137.469 rows=3 loops=3)
14	Buckets: 1024 Batches: 1 Memory Usage: 104kB
15	-> Parallel Seq Scan on reserved_ticket rt (cost=0.00..48364.67 rows=3 width=14) (actual time=50.377..137.416 rows=3 loop...
16	Filter: ((citizen_id)::text = 'AAAAA0000'::text)
17	Rows Removed by Filter: 969896
18	Planning Time: 0.503 ms
19	Execution Time: 255.826 ms

圖 14: Query plan before using index



	QUERY PLAN text
1	GroupAggregate (cost=4.89..57.63 rows=4 width=26) (actual time=2.738..2.739 rows=1 loops=1)
2	Group Key: nr.citizen_id
3	-> Nested Loop Left Join (cost=4.89..57.35 rows=32 width=18) (actual time=1.990..2.719 rows=32 loops=1)
4	Join Filter: ((nr.citizen_id)::text = (rt.citizen_id)::text)
5	-> Bitmap Heap Scan on non_reserved_ticket nr (cost=4.46..20.28 rows=4 width=14) (actual time=1.871..2.536 rows=4 loops=1)
6	Recheck Cond: ((citizen_id)::text = 'AAAAA0000'::text)
7	Heap Blocks: exact=4
8	-> Bitmap Index Scan on idx_nr_citizen_id (cost=0.00..4.46 rows=4 width=0) (actual time=1.043..1.044 rows=4 loops=1)
9	Index Cond: ((citizen_id)::text = 'AAAAA0000'::text)
10	-> Materialize (cost=0.43..36.61 rows=8 width=14) (actual time=0.029..0.042 rows=8 loops=4)
11	-> Index Scan using idx_rt_citizen_id on reserved_ticket rt (cost=0.43..36.57 rows=8 width=14) (actual time=0.103..0.153 rows=8 loop...
12	Index Cond: ((citizen_id)::text = 'AAAAA0000'::text)
13	Planning Time: 2.095 ms
14	Execution Time: 2.870 ms

圖 15: Query plan after using index

Using Index	Total Cost	Execution Time
No	65938.15	210ms
Yes	57.63	65ms

表 4: Comparison of Using Indexes or not

圖 14 和圖 15 分別是使用 index 前和後的 query plan，接著去執行這段查詢並記錄實際的執行時間，把結果簡單整理成表 4。可以觀察到使用 index 可以有效的減少 estimated total cost 並且實際執行時間也少了許多。

4

(a)

表 5 是三種方式所需要的 I/O 次數以及所需的時間。

Algorithm	cost	Total time
Stupid	$M + mN = 20000005000$ I/Os	556 hours
Single-block	$M + MN = 200005000$ I/Os	5.6 hours
Multi-block	$M + \lceil \frac{M}{B-2} \rceil N = 2085000$ I/Os	208.5 seconds

表 5: Comparison of different algorithm

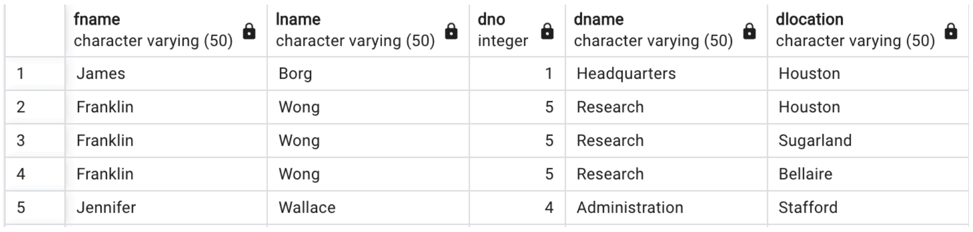
(b)

Stupid Nested Loop Join 的方法每一個 tuple 都要去對另一個表的 page 做一次 I/O，所以要做很多次 I/O，而 Single-block 則是每個 page 和另一個表的 page 做 I/O，所以會花費較少時間，Multi-block 跟 Single-block 比起來會同時用很多 buffer，所以時間會更近一步減少。

5

(a)

這句 SQL 是用來查詢每個員工的 `fname` 和 `lname`、他們所屬部門的編號及名稱以及該部門的位置。回傳的結果會顯示員工姓名、部門編號、部門名稱和部門的位置。



	fname character varying (50)	lname character varying (50)	dno integer	dname character varying (50)	dlocation character varying (50)
1	James	Borg	1	Headquarters	Houston
2	Franklin	Wong	5	Research	Houston
3	Franklin	Wong	5	Research	Sugarland
4	Franklin	Wong	5	Research	Bellaire
5	Jennifer	Wallace	4	Administration	Stafford

圖 16: Result of query

(b)

畫出來的樹狀圖如如 17 所示。

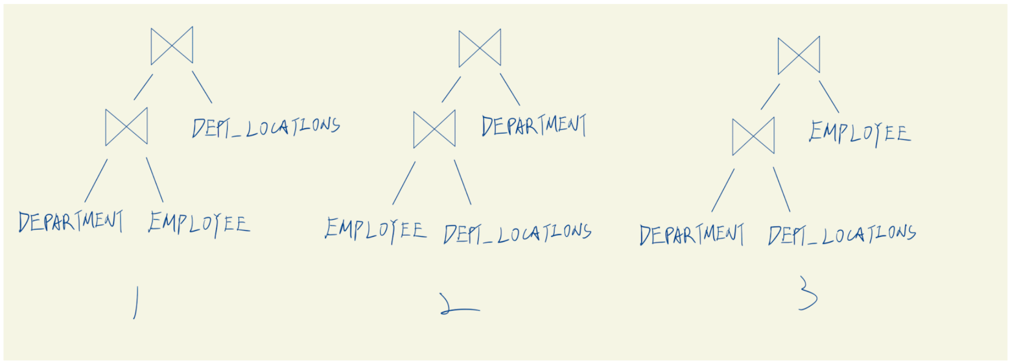
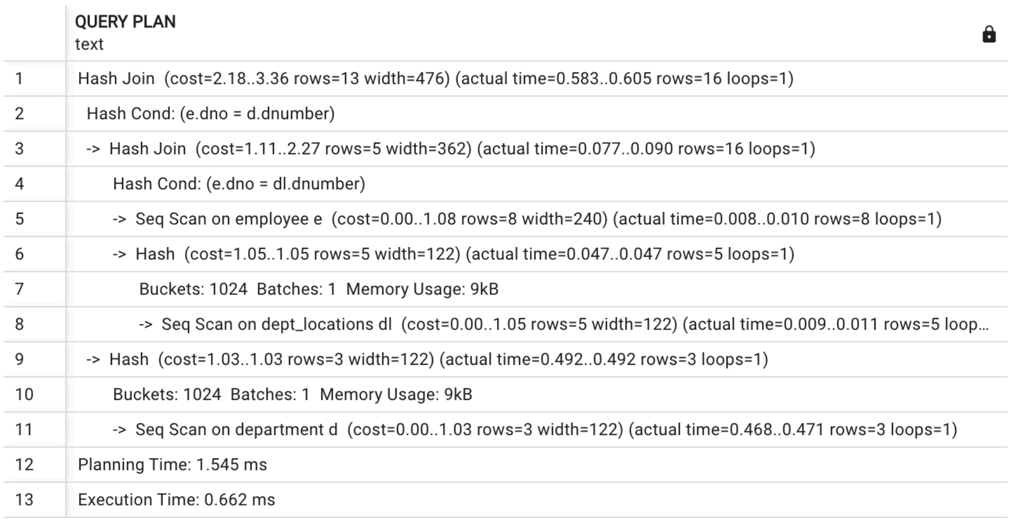


圖 17: 樹狀圖

(c)

RDBMS 執行的 query plan 如圖 18 所示，根據 query plan 可以看到 RDBMS 執行 join 的方式是第 2 種。RDBMS 會這樣建構的原因是因為 `DEPT_LOCATIONS` 和 `DEPARTMENT` 的資料表比較小，小的表會優先構建 Hash Table，這樣可以過濾掉更多資料，讓查詢更快。



	QUERY PLAN text
1	Hash Join (cost=2.18..3.36 rows=13 width=476) (actual time=0.583..0.605 rows=16 loops=1)
2	Hash Cond: (e.dno = d.dnumber)
3	-> Hash Join (cost=1.11..2.27 rows=5 width=362) (actual time=0.077..0.090 rows=16 loops=1)
4	Hash Cond: (e.dno = dl.dnumber)
5	-> Seq Scan on employee e (cost=0.00..1.08 rows=8 width=240) (actual time=0.008..0.010 rows=8 loops=1)
6	-> Hash (cost=1.05..1.05 rows=5 width=122) (actual time=0.047..0.047 rows=5 loops=1)
7	Buckets: 1024 Batches: 1 Memory Usage: 9kB
8	-> Seq Scan on dept_locations dl (cost=0.00..1.05 rows=5 width=122) (actual time=0.009..0.011 rows=5 loop...
9	-> Hash (cost=1.03..1.03 rows=3 width=122) (actual time=0.492..0.492 rows=3 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Seq Scan on department d (cost=0.00..1.03 rows=3 width=122) (actual time=0.468..0.471 rows=3 loops=1)
12	Planning Time: 1.545 ms
13	Execution Time: 0.662 ms

圖 18: Query plan

(d)

執行的 query plan 結果和前一小題的一樣，一樣的原因是 SQL 是 declarative 的語言，RDBMS 會選擇如何執行。

6

(a)

以下為使用 Python 實作的 hash join 演算法，表 6 則表示在餘數為 60、70 和 80 的 bucket 裡，各有幾筆 RESERVED_TICKET 資料。

```python
import psycopg2
import pandas as pd
import time

host = "localhost"
dbname = "THSR_database"
user = "postgres"
password = "1234"

conn = psycopg2.connect(
    host=host,
    dbname=dbname,
    user=user,
    password=password
)

cur = conn.cursor()

# Step 1
cur.execute("""
    SELECT arrive_station_id, citizen_id
    FROM reserved_ticket
    WHERE travel_date = '2023-08-01' AND depart_station_id <= 1000
""")
reserved_ticket_rows = cur.fetchall()

reserved_ticket_df = pd.DataFrame(reserved_ticket_rows, columns=["arrive_station_id",
↪   "citizen_id"])
print(f"Reserved Ticket records: {len(reserved_ticket_df)}")

# Step 2: Hash function
def hash_function(citizen_id):
    return sum(ord(c) for c in citizen_id) % 100

buckets = {i: [] for i in range(100)}
```

```python

# Put reserved_ticket data into buckets based on the hash function
for index, row in reserved_ticket_df.iterrows():
    citizen_id = row["citizen_id"]
    bucket_index = hash_function(citizen_id)
    buckets[bucket_index].append({"arrive_station_id": row["arrive_station_id"], "citizen_id": citizen_id, "name": None})


# Step 3
cur.execute("""
    SELECT citizen_id, name
    FROM member
    WHERE name LIKE '% A%'
""")
member_rows = cur.fetchall()

member_df = pd.DataFrame(member_rows, columns=["citizen_id", "name"])
print(f"Member records: {len(member_df)}")

# Step 4: Hash and join member data
start_hash_join = time.time()
for index, row in member_df.iterrows():
    citizen_id = row["citizen_id"]
    name = row["name"]
    bucket_index = hash_function(citizen_id)

    # Search for matching reserved_ticket data
    for reserved_ticket in buckets[bucket_index]:
        if reserved_ticket["citizen_id"] == citizen_id:
            reserved_ticket["name"] = name  # Fill the name column
end_hash_join = time.time()

# Step 5
result = []
for bucket in buckets.values():
    for reserved_ticket in bucket:
        if reserved_ticket["name"] is not None:  # Only output records with a filled name
            result.append((reserved_ticket["arrive_station_id"], reserved_ticket["name"]))
```

```python

result_df = pd.DataFrame(result, columns=["arrive_station_id", "name"])

print(f"Hash join execution time: {end_hash_join - start_hash_join:.6f} seconds")
print(result_df)

cur.close()
conn.close()
```

Bucket	Record Count
60	291
70	27
80	0

表 6: Record counts in buckets

(b)

下面為使用暴力 join 演算法的 Python 程式碼。

```python
import psycopg2
import pandas as pd
import time

host = "localhost"
dbname = "THSR_database"
user = "postgres"
password = "1234"

conn = psycopg2.connect(
    host=host,
    dbname=dbname,
    user=user,
    password=password
)

cur = conn.cursor()

# Step 1
cur.execute("""
    SELECT arrive_station_id, citizen_id
    FROM reserved_ticket
    WHERE travel_date = '2023-08-01' AND depart_station_id <= 1000
""")
reserved_ticket_rows = cur.fetchall()

reserved_ticket_df = pd.DataFrame(reserved_ticket_rows, columns=["arrive_station_id",
 ↪  "citizen_id"])

# Step 3
cur.execute("""
    SELECT citizen_id, name
    FROM member
    WHERE name LIKE '% A%'
""")
member_rows = cur.fetchall()

member_df = pd.DataFrame(member_rows, columns=["citizen_id", "name"])

# Step 4: Brute-force join using nested loops
result = []

reserved_ticket_array = reserved_ticket_df.to_numpy()
member_array = member_df.to_numpy()

# Nested loops to perform the join
start_time = time.time()
for member in member_array:
    matches = reserved_ticket_array[:, 1] == member[0]  # Compare citizen_id
    for idx, match in enumerate(matches):
        if match:
            result.append((reserved_ticket_array[idx, 0], member[1]))  #
             ↪  arrive_station_id, name

brute_force_time = time.time() - start_time

# Step 5
result_df = pd.DataFrame(result, columns=["arrive_station_id", "name"])

print(f"Brute-force join execution time: {brute_force_time:.6f} seconds")
print(result_df)

cur.close()
```

```
conn.close()
```

(c)

表 7 為兩種演算法在實際執行 join 時所需要花費的時間，可以看到暴力 join 的執行時間比 hash 多很多，因為 join 要跑雙層回圈，複雜度是 $\mathcal{O}(n \times m)$，在資料量很龐大的時候暴力算法就要算很久。而 hash join 相較於暴力 join 的缺點是要花費更多的儲存空間。

Algorithm	**Execution time**
Hash join	0.568619 seconds
Brute-force	15.090085 seconds

表 7: Comparison of different algorithm