Smart Flight Searching

1st Yutong Lei
College of Sci and Engineering
University of Minnesota Twin Cities
Minneapolis, Minnesota
Lei00025@umn.edu

2nd Dongwei Pan
College of Sci and Engineering
University of Minnesota Twin Cities
Minneapolis, Minnesota
Pan00179@umn.edu

3th Yinzhao Wang
College of Liberal Arts
University of Minnesota, Twin Cities
Minneapolis, Minnesota
wan00232@umn.edu

4rd Hexuan Zhang
College of Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, Minnesota
zhan7790@umn.edu

***Abstract*—In this paper, we talk about our Smart Flight Searching algorithm. We build our own database and accomplish searching the best flights according to departure and destination airports with their corresponding time, adjusting with respect to the preference of traveling time or arriving time, or the necessity to avoid transferring. In Section II, we talk about the data we have and the ER-diagram we designed. In Section III, the focus is on the method we use, the final one and the failed one. In Section V, the result of testing are shown and analyzed. And in Section VI, we discuss the possibility of improving and expanding our model.**

***Index Terms*—Database Management System, SQL, Flight Searching**

I. Introduction

Since the invention of airplane, it has became one of the most important transportation ways in our lives. According to ICAO, in the year 2019 alone, there were 38.3 million flights all over the world with 4.5 billion passengers, which is 12 million passengers per day and about 1 million passengers in the sky any moment.[**b1**] Without any doubt, finding a convenient flight is of high value, and has been studied for a long time. Each airline company has their own searching tools, and there are multiple independent companies combine and search the best flights possible. In our project, we design our own database and algorithms with much smaller and limited data but similar effect. We have chosen 9 large airports and some small airports for illustration the usefulness of our model. Also, it has been proven to be difficult to have real-time data. Therefore, we will use collected data of year 2021. Our data points are limited but we expect our algorithms to be effective on larger datasets as well.

II. Data Description

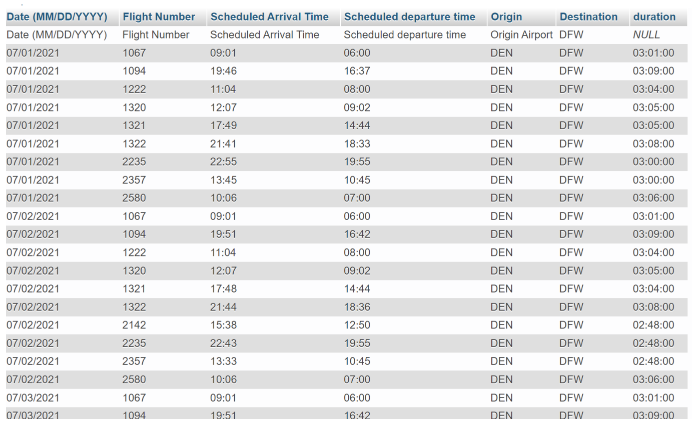
A. Data Sources

We downloaded the data from Bureau of Transportation Statistics. BTS provides updated departure and arrival data of airports in the US, with detailed flight information such as flight number, destination, and airline name. We also have the data from OpenSky historical database. This dataset contains some historical live flight data, and it could be complementary to the BTS dataset.

We dropped some columns that we did not use, such as the size, the model of the plane, the terminal they depart or arrive. We combined the small tables together. We also calculated the duration of the flights by subtracting the departure time from the arrival time. Finally, our table has the following columns: date, flight numbers, scheduled departure time, scheduled arrival time, origin, destination, duration. See the snapshot of the table in Fig. 1

B. ER-Diagram

We constructed our ER-Diagram like shown in 2, for Airlines, we have key Name, and avg price. For flights, we have key Number, other attributes



Date (MM/DD/YYYY)	Flight Number	Scheduled Arrival Time	Scheduled departure time	Origin	Destination	duration
Date (MM/DD/YYYY)	Flight Number	Scheduled Arrival Time	Scheduled departure time	Origin Airport	DFW	*NULL*
07/01/2021	1067	09:01	06:00	DEN	DFW	03:01:00
07/01/2021	1094	19:46	16:37	DEN	DFW	03:09:00
07/01/2021	1222	11:04	08:00	DEN	DFW	03:04:00
07/01/2021	1320	12:07	09:02	DEN	DFW	03:05:00
07/01/2021	1321	17:49	14:44	DEN	DFW	03:05:00
07/01/2021	1322	21:41	18:33	DEN	DFW	03:08:00
07/01/2021	2235	22:55	19:55	DEN	DFW	03:00:00
07/01/2021	2357	13:45	10:45	DEN	DFW	03:00:00
07/01/2021	2580	10:06	07:00	DEN	DFW	03:06:00
07/02/2021	1067	09:01	06:00	DEN	DFW	03:01:00
07/02/2021	1094	19:51	16:42	DEN	DFW	03:09:00
07/02/2021	1222	11:04	08:00	DEN	DFW	03:04:00
07/02/2021	1320	12:07	09:02	DEN	DFW	03:05:00
07/02/2021	1321	17:48	14:44	DEN	DFW	03:04:00
07/02/2021	1322	21:44	18:36	DEN	DFW	03:08:00
07/02/2021	2142	15:38	12:50	DEN	DFW	02:48:00
07/02/2021	2235	22:43	19:55	DEN	DFW	02:48:00
07/02/2021	2357	13:33	10:45	DEN	DFW	02:48:00
07/02/2021	2580	10:06	07:00	DEN	DFW	03:06:00
07/03/2021	1067	09:01	06:00	DEN	DFW	03:01:00
07/03/2021	1094	19:51	16:42	DEN	DFW	03:09:00

Fig. 1. Large table snapshot

like price, duration, and date. Airports, we have key Name, and other attributes like city they are in, whether they are large, longitude and latitude, and average price of tickets from the airport. Flights belong to some Airline, and Flights arrive or depart at Airports.

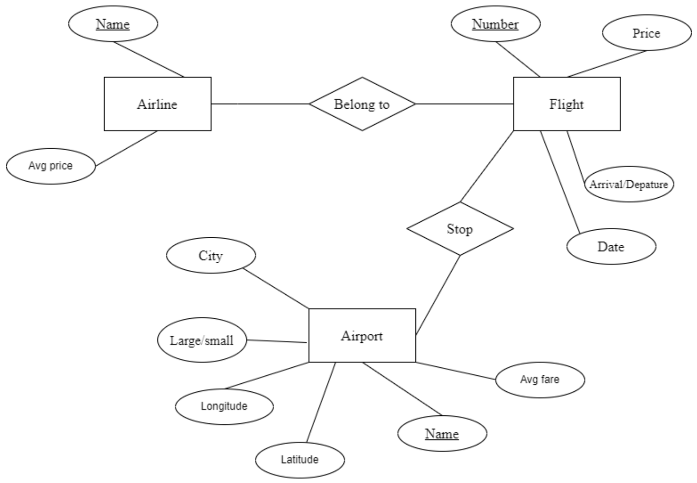


Fig. 2. ER-Diagram

III. Preliminary

A. Direct Flights Search

The searching of direct flights are straightforward and easy, all we need to do is select the departure and destination airports and sort them according to traveling time or arriving time. When we search each flight of flights with transfer, we used this same algorithm.

B. Transfer Flights Search

For most of the time, flights between two small airports are not always there and convenient, taking a transfer is much faster and more reasonable. However, with the consideration of transferring, searching the best flight becomes much more tedious. We need to look at the flights to every other airports and then from each airports, we check the flights to the destination airport. Moreover, adding more airports to our database in the future will increase the complexity too much. Such method is too costly and unrealistic. Imagine schedule a flight which transfers in an airport that nobody knows where it is. With such consideration in mind, we had two different methods and chosen the latter as our final one.

1) "Anchor Point Airport" Method: Now that we know it is troublesome to consider small airports, the most obvious solution to reduce searching time is to link small airports to their closest large airports, which we call "*Anchor*". With this idea in mind, we split the map of United States in grid, with respect to longitude and latitude. Within each grid, we select the airport with the most flights available as the *Anchor* airport. For any other airports within the grid, if a person wants to transfer, they will fly to the *Anchor* of the departure airport, transfer and fly to the *Anchor* of the destination airport, transfer and fly to the destination airport. This sound more hideous than it actually is, because if there is a direct flight which is most likely to be faster, this transferring flight will not be the best choice. If there is none, taking this kind of transfer would be normal for people living in a small town.

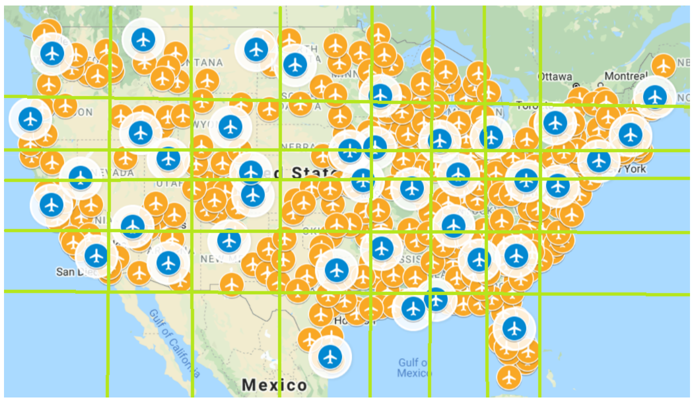


Fig. 3. Selected Anchor Points

One of the advantages of this method is that we reduce the total number of airports that we need to check to a very limited number. When we are searching for a transfer, it will always be between two *Anchor* airports, so we can have a separate table that is much smaller than all the airports combined.

However, there are some drawbacks of this method. We only consider the geometric features of the airports in grid. The populations are different across the United States, and the distribution of cities is not uniform. The cities on the west coast and the east coast of United States have much more people living in and much more airports in there. Moreover, with *Anchor* airports, most of the flights that need transfer will have to transfer twice, which is not the most comfortable traveling plan. And this is the the major reason we decided to change to the following method, with one transfer.

2) Large Airports Transfer: As mentioned above, our goal is not only to reduce searching time, but also make the flight plan as comfortable as we can. Thus, in this method, we limit our transfers to be at most once, and be in a *large* airport. With this method, instead of checking all the airports for availability of transferring, we limited the count to 9, as we will discuss in the following paragraph. But this method will still look through the entire table and will not be as time saving as the *Anchor point airport* method.

We counted the number of flights from and to each airports, and sort them descending, as shown in I. Nonetheless, the distribution of the top airports are not as optimal as it would be, and we take some of the idea of *Anchor* method and try to cover the whole United States evenly. From the airports shown in I, we decided to delete EWR, LAS, and BOS because they are geometrically close to a larger airport and will not be very useful for our flight search, and we have added SEA to the *large* airport list and expect the transfer recommendation be more reasonable.

With this 9 *large* airports, when we want to search for a flight with transfer, we will search for the first flight from the departure airport to these 9 *large* airports, we then give a gap of 1 hour for transferring, and then we search for the first flight to the destination airport from each of these 9 *large* airports after the corresponding first flight.

Airport	City	No. of Flights	Coordinates
ATL	Atlanta	36514	(-84.4,33.6)
LAX	Los Angeles	14394	(-118.4,33.9)
ORD	Chicago	20347	(-87.9,41.9)
DFW	Dallas	29113	(-97.0,32.8)
DEN	Denver	15642	(-104.6,39.8)
JFK	New York City	22802	(-73.7,40.6)
SFO	San Francisco	13975	(-112.3,37.6)
MCO	Orlando	13836	(-81.3,28.4)
SEA	Seattle	10049	(-122.3,47.4)

TABLE I
SELECTED AIRPORTS

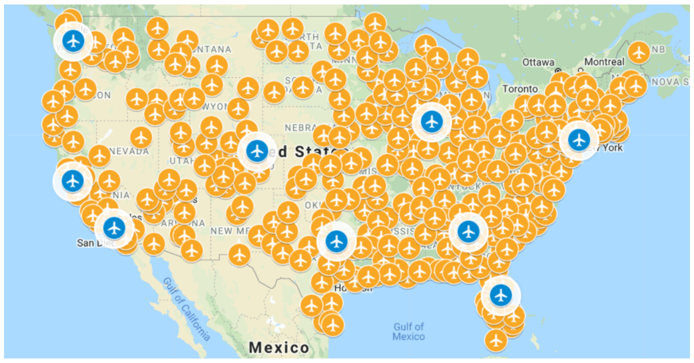


Fig. 4. Chosen Large Airports

As shown in Fig. 8, the *large* airports are somewhat evenly distributed all over America, except for Alaska.But if we were to schedule a flight with transfer to or from Alaska, we will add the largest airport in Alaska as our 10th *large* airport. When we add more airports into the database, these 9 will still be very helpful for our flight search.

IV. Algorithm and Implementation

When a flight query is done, most of the flight search engines return results based on some specific ranking criteria, such as cheapest total price, shortest total travel time, least transfer, and so on. To simulate a real world experience, we implemented two rankings, earliest arrival and shortest total travel time, which are doable based on our data set. In this section, we will discuss in detail about our tool-kits and query implementation.

A. Tool-kits

Implementation of the project involved following tools:

1) Mysql (For database storage and query)
2) XAMPP (For acessing local Mysql serve)

3) Python (For data wrangling and implementing extra features incapable for Mysql)

B. Earliest arrival search

```
Algorithm 1 No transfer, Earliest arrival
Require: No transfer node $TR$.
1: Set-up:
    1) Define start point $S$ and destination $D$.
    2) Set list of query result $EA$.
    3) Function Query(S,D) returns all flights from $S$ to $D$.
2: EA $\longleftarrow$ Query(S,D)
3: return(rank(EA, 10))
```

In this section, we will discuss two procedures involves in earliest arrival search - No transfer and Multiple transfers.

In purpose of reducing complexity, our project mainly implemented case of no transfer and two or less transfer. However, multiple transfers beyond two shares similar logic as two transfers though with exponentially increasing complexity, and this will be discussed in following.

1) No transform case: In scenario of no transfer, query is straightforward. We only need to select all direct flights between two cities and rank by arrival time. Following is a detailed algorithm.

2) Multiple transfers (less or equal 2): To query flights with multiple transfers, we need compare all possible combinations of flights from start to destination with required number of transfers.

C. Shortest travel time

Technically, shortest travel time problem is isomorphic except for only difference in ranking criteria. To switch from earliest arrival to shortest travel time, we need to change ranking in arrival time to ranking in travel duration.

Comparing to earliest arrival problem, where there is no shortcut but a comprehensive query and comparison among all possible flights, shortest travel time problem might be able to solve in a more efficient way. We will discuss some possible approaches in part V

```
Algorithm 2 Two or less transfer, Earliest arrival
Require: Transfer nodes $TR \in$ {all large airports}.
1: Set-up:
    1) Assume $G$ number of transferable nodes.
    2) Define start point $S$ and destination $D$.
    3) Flight recorder $E_{Sq}[G]$, $E_{Sqp}[G,G]$, $E_{SqpD}[G,G,G]$
    4) $q$ is a choice of first transfer, and $p$ is a choice of second transfer.
    5) Function Query($TR_q, TR_p$) returns all flights from $TR_q$ to $TR_p$, such that $Arrival(TR_q) - Depature(TR_p) >$ 1 hour
    6) Function $rank(Query, 1)$ returns earliest arrival flight.
2: For each $p$, $(Query(S, TR_q)) \longrightarrow E_{Sq}[G]$
3: Given $p$ and for each $q$, $(Query(TR_q, TR_p)) \longrightarrow E_{qp}[G]$
4: Given $p, q$, $(QUER(TR_p, D)) \longrightarrow E_{SqpD}[G,G,G]$
5: Return 10 $SqpD$ such that corresponding $E_{SqpD}[G,G,G]$ are 10 smallest.
```

1) Implementation: Our project implemented the above algorithm in MySQL and include several features majorly realized through Python. A detailed description of functions and implementation is as follows:

1) Prepossess the flight table by join with all possible transferring airport. We get a list of tables corresponding to each airports and containing all flights in the airport.
2) Create a simple user interface allowing user to specify transfer points.
3) Create a flexible non-transfer query framework, which enables filling in departure and destination airports. The idea is to select columns from the inner join of departure and destination tables. This is exactly implementation of *Algorithm 1*
4) Use the query in 3) to as function Query in *Algorithm 2*. Then write loops in Python that query through all possible combinations.
5) Use result of each loop as subqueries and select required columns with rows order by

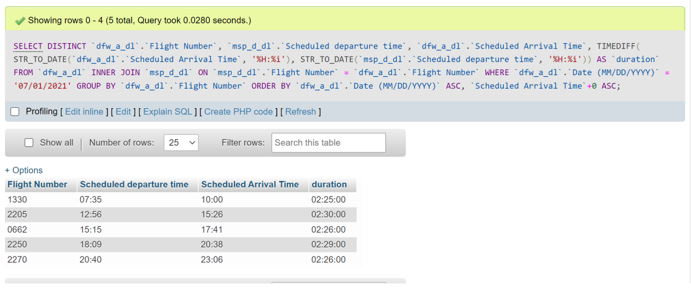
some criteria.

Most of above procedures are direct translation of *Algorithm 1* and *Algorithm 2* and self-explained. However, step 1) might need more explanation on its motivation.

If we perform all search on full flight table, we need to use self join each time we search a flight. Since the full flight table is large, with more than ten thousand rows, looped and nested self joins are expensive. To handle the problem, we separated tables, each with around hundreds rows. Then times is significantly saved.

2) Example: We demonstrate our work through an example of flight search between Minneapolis and Dallas.

i) Non-transfer case Figure 5 shows the 4 results we obtained through from the above code. In total, it took around 0.0280 seconds for searching.


Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]
Show all
Number of rows: 25
Filter rows: Search this table
+ Options

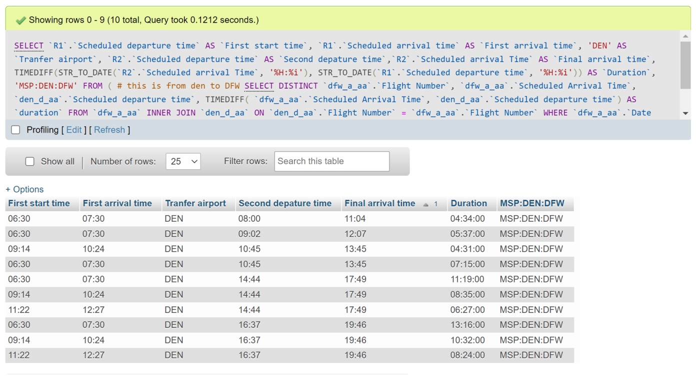
Showing rows 0 - 4 (5 total, Query took 0.0280 seconds.)

```
SELECT DISTINCT `dfw_a_dl`.`Flight Number`, `msp_d_dl`.`Scheduled departure time`, `dfw_a_dl`.`Scheduled Arrival Time`, TIMEDIFF(
STR_TO_DATE(`dfw_a_dl`.`Scheduled Arrival Time`, '%H:%i'), STR_TO_DATE(`msp_d_dl`.`Scheduled departure time`, '%H:%i')) AS `duration`
FROM `dfw_a_dl` INNER JOIN `msp_d_dl` ON `msp_d_dl`.`Flight Number` = `dfw_a_dl`.`Flight Number` WHERE `dfw_a_dl`.`Date (MM/DD/YYYY)` =
'07/01/2021' GROUP BY `dfw_a_dl`.`Flight Number` ORDER BY `dfw_a_dl`.`Date (MM/DD/YYYY)` ASC, `Scheduled Arrival Time`+0 ASC;
```

Flight Number	Scheduled departure time	Scheduled Arrival Time	duration
1330	07:35	10:00	02:25:00
2205	12:56	15:26	02:30:00
0662	15:15	17:41	02:26:00
2250	18:09	20:38	02:29:00
2270	20:40	23:06	02:26:00

Fig. 5. MSP-DFW

ii) One transfer case For simpler demonstration, we pick two transfer candidates, Chicago (ORD) and Denver (DEN). Figure 6 and Figure 7 demonstrated respective search result.

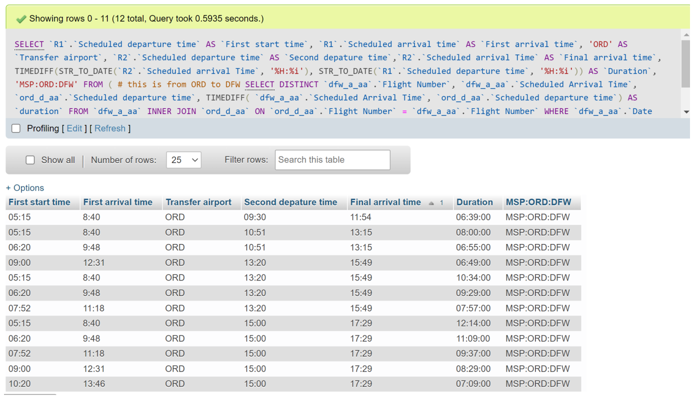

Profiling [Edit] [Refresh]
Show all
Number of rows: 25
Filter rows: Search this table
+ Options

Showing rows 0 - 9 (10 total, Query took 0.1212 seconds.)

```
SELECT `R1`.`Scheduled departure time` AS `First start time`, `R1`.`Scheduled arrival time` AS `First arrival time`, 'DEN' AS
`Tranfer airport`, `R2`.`Scheduled departure time` AS `Second depature time`,`R2`.`Scheduled arrival Time` AS `Final arrival time`,
TIMEDIFF(STR_TO_DATE(`R2`.`Scheduled arrival Time`, '%H:%i'), STR_TO_DATE(`R1`.`Scheduled departure time`, '%H:%i')) AS `Duration`,
'MSP:DEN:DFW' FROM ( # this is from den to DFW SELECT DISTINCT `dfw_a_aa`.`Flight Number`, `dfw_a_aa`.`Scheduled Arrival Time`,
`den_d_aa`.`Scheduled departure time`, TIMEDIFF( `dfw_a_aa`.`Scheduled Arrival Time`, `den_d_aa`.`Scheduled departure time`) AS
`duration` FROM `dfw_a_aa` INNER JOIN `den_d_aa` ON `den_d_aa`.`Flight Number` = `dfw_a_aa`.`Flight Number` WHERE `dfw_a_aa`.`Date
```

First start time	First arrival time	Tranfer airport	Second depature time	Final arrival time ▲ 1	Duration	MSP:DEN:DFW
06:30	07:30	DEN	08:00	11:04	04:34:00	MSP:DEN:DFW
06:30	07:30	DEN	09:02	12:07	05:37:00	MSP:DEN:DFW
09:14	10:24	DEN	10:45	13:45	04:31:00	MSP:DEN:DFW
06:30	07:30	DEN	10:45	13:45	07:15:00	MSP:DEN:DFW
06:30	07:30	DEN	14:44	17:49	11:19:00	MSP:DEN:DFW
09:14	10:24	DEN	14:44	17:49	08:35:00	MSP:DEN:DFW
11:22	12:27	DEN	14:44	17:49	06:27:00	MSP:DEN:DFW
06:30	07:30	DEN	16:37	19:46	13:16:00	MSP:DEN:DFW
09:14	10:24	DEN	16:37	19:46	10:32:00	MSP:DEN:DFW
11:22	12:27	DEN	16:37	19:46	08:24:00	MSP:DEN:DFW

Fig. 6. MSP-DEN-DFW


Profiling [Edit] [Refresh]
Show all
Number of rows: 25
Filter rows: Search this table
+ Options

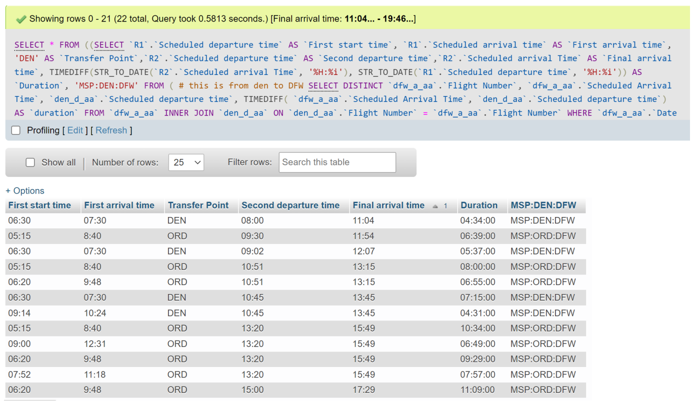
Showing rows 0 - 11 (12 total, Query took 0.5935 seconds.)

```
SELECT `R1`.`Scheduled departure time` AS `First start time`, `R1`.`Scheduled arrival time` AS `First arrival time`, 'ORD' AS
`Transfer airport`, `R2`.`Scheduled departure time` AS `Second depature time`,`R2`.`Scheduled arrival Time` AS `Final arrival time`,
TIMEDIFF(STR_TO_DATE(`R2`.`Scheduled arrival Time`, '%H:%i'), STR_TO_DATE(`R1`.`Scheduled departure time`, '%H:%i')) AS `Duration`,
'MSP:ORD:DFW' FROM ( # this is from ORD to DFW SELECT DISTINCT `dfw_a_aa`.`Flight Number`, `dfw_a_aa`.`Scheduled Arrival Time`,
`ord_d_aa`.`Scheduled departure time`, TIMEDIFF( `dfw_a_aa`.`Scheduled Arrival Time`, `ord_d_aa`.`Scheduled departure time`) AS
`duration` FROM `dfw_a_aa` INNER JOIN `ord_d_aa` ON `ord_d_aa`.`Flight Number` = `dfw_a_aa`.`Flight Number` WHERE `dfw_a_aa`.`Date
```

First start time	First arrival time	Transfer airport	Second depature time	Final arrival time ▲ 1	Duration	MSP:ORD:DFW
05:15	8:40	ORD	09:30	11:54	06:39:00	MSP:ORD:DFW
05:15	8:40	ORD	10:51	13:15	08:00:00	MSP:ORD:DFW
06:20	9:48	ORD	10:51	13:15	06:55:00	MSP:ORD:DFW
09:00	12:31	ORD	13:20	15:49	06:49:00	MSP:ORD:DFW
05:15	8:40	ORD	13:20	15:49	10:34:00	MSP:ORD:DFW
06:20	9:48	ORD	13:20	15:49	09:29:00	MSP:ORD:DFW
07:52	11:18	ORD	13:20	15:49	07:57:00	MSP:ORD:DFW
05:15	8:40	ORD	15:00	17:29	12:14:00	MSP:ORD:DFW
06:20	9:48	ORD	15:00	17:29	11:09:00	MSP:ORD:DFW
07:52	11:18	ORD	15:00	17:29	09:37:00	MSP:ORD:DFW
09:00	12:31	ORD	15:00	17:29	08:29:00	MSP:ORD:DFW
10:20	13:46	ORD	15:00	17:29	07:09:00	MSP:ORD:DFW

Fig. 7. MSP-ORD-DFW

iii) Combined result Figure 8 shows return of query combined result from figure 7 and figure 8, ranked by the final arrival time. We could observe that it took around 0.58 seconds on performing the search, which is an acceptable time.


Profiling [Edit] [Refresh]
Show all
Number of rows: 25
Filter rows: Search this table
+ Options

Showing rows 0 - 21 (22 total, Query took 0.5813 seconds.) [Final arrival time: **11:04... - 19:46...**]

```
SELECT * FROM ((SELECT `R1`.`Scheduled departure time` AS `First start time`, `R1`.`Scheduled arrival time` AS `First arrival time`,
'DEN' AS `Transfer Point`,`R2`.`Scheduled departure time` AS `Second departure time`,`R2`.`Scheduled arrival Time` AS `Final arrival
time`, TIMEDIFF(STR_TO_DATE(`R2`.`Scheduled arrival Time`, '%H:%i'), STR_TO_DATE(`R1`.`Scheduled departure time`, '%H:%i')) AS
`Duration`, 'MSP:DEN:DFW' FROM ( # this is from den to DFW SELECT DISTINCT `dfw_a_aa`.`Flight Number`, `dfw_a_aa`.`Scheduled Arrival
Time`, `den_d_aa`.`Scheduled departure time`, TIMEDIFF( `dfw_a_aa`.`Scheduled Arrival Time`, `den_d_aa`.`Scheduled departure time`)
AS `duration` FROM `dfw_a_aa` INNER JOIN `den_d_aa` ON `den_d_aa`.`Flight Number` = `dfw_a_aa`.`Flight Number` WHERE `dfw_a_aa`.`Date
```

First start time	First arrival time	Transfer Point	Second departure time	Final arrival time ▲ 1	Duration	MSP:DEN:DFW
06:30	07:30	DEN	08:00	11:04	04:34:00	MSP:DEN:DFW
05:15	8:40	ORD	09:30	11:54	06:39:00	MSP:ORD:DFW
06:30	07:30	DEN	09:02	12:07	05:37:00	MSP:DEN:DFW
05:15	8:40	ORD	10:51	13:15	08:00:00	MSP:ORD:DFW
06:20	9:48	ORD	10:51	13:15	06:55:00	MSP:ORD:DFW
06:30	07:30	DEN	10:45	13:45	07:15:00	MSP:DEN:DFW
09:14	10:24	DEN	10:45	13:45	04:31:00	MSP:DEN:DFW
05:15	8:40	ORD	13:20	15:49	10:34:00	MSP:ORD:DFW
09:00	12:31	ORD	13:20	15:49	06:49:00	MSP:ORD:DFW
06:20	9:48	ORD	13:20	15:49	09:29:00	MSP:ORD:DFW
07:52	11:18	ORD	13:20	15:49	07:57:00	MSP:ORD:DFW
06:20	9:48	ORD	15:00	17:29	11:09:00	MSP:ORD:DFW

Fig. 8. combined result

V. Results and Analysis

In this section, we will discuss about efficiency of query and possible ways of improvement. Basically, there are two ways to approach enhancement. One is to optimize query codes, and the other is to reform the problem into a new one with existing efficient solutions.

A. Indexing and query acceleration

To improve speed of code without change the basic structure, we could create a multi-column index on the date and flight number.

This index would accelerate the query on full table since the flights are randomly ordered. However, since we have already create sub-tables, with around a thousand rows, index acceleration would be as significant as expect.

B. *Reform the problem*

If we take a further inspect, earliest arrival problem could be solved in a backward searching way ,and shortest travel time problem is isomorphic to a shortest path problem, where travel time of each flight segment could be map to edge of a graph and each airport is just a node in the graph.

i) earliest arrival: Remember in the previous section, we basically performed a exhausting search for earliest arrival problem, which is definitely worst case comparing any possible algorithms. The drawback of high complexity could be relieved by backward search.

We instead start from destination and search a earliest arrival flight between itself and each possible transfer. Then, perform same procedure on each

ii) shortest travel time: In the case of shortest travel time ranking, we are lucky to have some existing and well-developed algorithm to efficiently deal with it. One most straightforward and immediate one is graph shortest path search algorithm.

One commonly used graph shortest path search algorithm is Dijkstra's algorithm. To perform the algorithm, we need to build up a weighted graph. One difficulty here is how to find a reasonable weight. A simple and straightforward idea is to use averaged flight time between two cities as weighted edge. However, this method ignore the variance among different flights. We inspected the data and found that there could be a variance up to around 40 minutes. Such high variance would significantly affect our flight ranking.

After doing some research, we found an improved solution, the time dependent shortest path problem. The procedure of solving this problem is similar to that of Dijikstra's algorithm. Most of difference come from that there is a time dependent weighted graph. For each time we move from one airport to another, graph weight will change depend on a given time.

If we want to implement this, it only requires add a python realized Dijkstra's algorithm with a weighted graph defined on quickest flight between two nodes and changing with movement from one transfer to another.

Two alternative ways discussed before could systematically decrease the complexity of queries.

VI. Discussion

A. *Further possibilities*

1) Price: Prices of tickets are generally hard to get since they are confidential information for companies. Furthermore, booking in a different time results in a different cost. Even though we have the average price for flight routes and airports, these cannot help much. Searching with prices requires more data and more complicated algorithm.

2) Database Size: In this project, we used one month of the data, flights of 3 airline companies, among 9 large airports and 9 smaller airports. In the future, to get full functionality, we could add more airports and more airline companies. For the time of the flights, it is possible for us to get real-time data by designing some applications that we will not discuss in this course.

B. *What we learned*

We were able to gain more experiences since we use some real data for built our database, drew a ER-diagram and came up with some queries and implemented them. Based on that point, we learned how to built some database for a program, how to use data for support a program. We also learned how to use some function to index some queries and how to implemented these queries by algorithms in the program. We not only learned something new but also review some important points in the class, also how to combine the theory and real program. Thus, these are really we learned during this project.

Acknowledgment

Data resources

- Bureau of Transportation Statistics, https://transtats.bts.gov/ONTIME/Arrivals.aspx
- Opensky, https://opensky-network.org/

Contribution of teammates

All four of us had group meeting biweekly to discuss the general algorithms and decided what to do in each step. As for more specific steps, Dongwei Pan downloaded and cleaned the data.

Hexuan Zhang wrote direct flight search SQL and decided to use Large Airport algorithm. Yinzhao Wang wrote the final transfer flight search algorithm and SQL. Yutong Lei wrote the Python interface and analyzed our results.

References

[1] ICAO, https://www.icao.int/annual-report-2019/Pages/the-world-of-air-transport-in-2019.aspx

[2] Zhao, L., Ohshima, T., Nagamochi, H. (2008, July). $A*$ Algorithm for the time-dependent shortest path problem.

[3] Ramakrishnan, Raghu and Gehrke, Johannes. Database Management Systems