# Flight On-Time and Delay Statistics Database

Khushmeet Shergill
*SEAS*
*SUNY Buffalo*
khushmee@buffalo.edu

Sai Sri Pavan Dandu
*SEAS*
*SUNY Buffalo*
sdandu@buffalo.edu

Mohit Ramna
*SEAS*
*SUNY Buffalo*
mohitram@buffalo.edu

***Abstract*—In this project related to Database Systems, we use production system flight data to demonstrate database system capabilities on PostgreSQL and create a web application that connects to the said database and run queries and display results.**

***Index Terms*—DMQL, PostgreSQL, Flask, Python**

## I. Problem Statement

Today we have a huge operational system governing airlines in the United States. Due to such huge system, error can bleed into the system, and they are unpredictable. Moreover, there are external factors that could affect the system. Such as flights being delayed due to weather or cancelled because of geopolitical situation or getting diverted due to a storm. Our aim with this project is to build a database system to hold the flight on-time and delay statistics and analyze the data for trends and patterns in delay and its reason. This will help airline administrators to plan according for any issue that can happen.

This problem requires database because of the sheer volume of data that is generated from these systems. For this project we are using 4 years' worth of data with around 80,000 rows. This is a subset of the data available, but even this subset can prove difficult to use in excel. We must bring whole data into system memory for querying which can be almost impossible for large amount of data. In such scenarios, database system helps to store and organize data efficiently for easy access and performing data operations. Moreover, database systems provide excellent backup features providing data redundancy and high availability, which excel does not.

## II. Target User

We expect our database system to be run by Federal Aviation Administration for performing aviation forecasts. It involves analyzing trend to predict future demands by designing statistical models. Data is processed to remove any sensitive information related to commercial jets or its passengers. We expect database to publicly available inside the organization on read privilege, while roles with certain update privilege is given to automated system to periodically insert data to database to keep it up to date. Only the administrator can modify the database structure to incorporate further attributes in the data. Another target user for such system are the researchers and students who want to use data to design and perform data analysis and predictive modeling. This data can be provided as a standalone database file.

## III. DB Integrity

DB integrity refers to the correctness and reliability of data. Integrity is preserved by a series of checks and rules that are executes during the creation of the relations. These checks make sure that no transaction in database reads to the data corruption. We have implemented entity, referential and domain integrity rules so that our database is consistent.

For the six relations present in the database we defined integer primary key, that uniquely identifies a row in the relation. This primary key is also used to establish referential integrity by defining relationships among the relations.

For domain integrity, we define domains and constraints on the attributes so that no invalid values are inserted into the database. For referential integrity, we define references between relations to signify how the rows are linked to other row in other relation. This makes sure that data is not duplicated across table and prevents incomplete data being returned from queries.

Here are the relation details, with all the integrity checks.

**State Relation** - This relation stores the state name of all the states in United States.

- ***id*** – unique state number
  *constraints*: primary key not null
- ***state_name*** – Name of the state
  *constraints*: not null unique

**City Relation** - This relation is used to store city names and its corresponding state_id. It tells in which US state the city is in.

- ***id*** – unique city number
  *constraints*: primary key not null
- ***city_name*** – Name of city
  *constraints*: not null
- ***state_id*** – ID of state
  *constraints*: foreign key references state(id) on delete cascade

**Airport_loc Relation** - Airport relation stores the details about an airport. This includes its name, its location in latitude and longitude.

- ***id*** – unique airport number
  *constraints*: primary key not null
- ***airport_name*** – Airport name
  *constraints*: not null unique
- ***latitude*** – Latitude of airport
  *constraints*: None

- ***longitude*** – Longitude of airport
  *constraints*: None

**Airport Relation** - Airport relation stores the details about an airport. This includes its IATA code, size of the airport, city and state it belongs to.

- ***id*** – unique airport number
  *constraints*: primary key not null
- ***iata*** – International Air Transportation Authority (IATA) code
  *constraints*: not null unique
- ***type*** – Type of airport by size
  *constraints*: Check in (large, medium, small, closed)
- ***city_id*** – ID of the city where airport is located
  *constraints*: foreign key references city(id) on delete cascade on update cascade
- ***state_id*** – ID of the state where airport is located
  *constraints*: foreign key references state(id) on delete cascade on update cascade

**Airline Relation** - Airline relation stores attributes of an airline. This includes its name, IATA code, ICAO code etc. this relation has no foreign keys.

- ***id*** – Unique id for airline
  *constraints*: primary key not null
- ***airline_name*** – Name of the airline
  *constraints*: not null unique
- ***iata*** – IATA code for airline
  *constraints*: None
- ***icao*** – Internation civil aviation organization code
  *constraints*: None
- ***callsign*** – Aircraft callsign
  *constraints*: None

**Flight Relation** - Flights relation stores the flight statistics for a given airline on a given airport, for every month of the year from 2018 to 2021. Statistics include the number of aircrafts that were delayed due to variety of reasons like, security, or aviation related delays or weather-related delays. Additionally, it stores the number of minutes of the delay that happened due to the reasons mentioned above. Every tuple is connected to an airline and airport relation, that informs which airline this statistics belong to and where the delays happened.

- ***id*** – Unique identifier for each entry
  *constraints*: primary key not null
- ***arr_flights*** – Number of flights which arrived at the airport
  *constraints*: None
- ***arr_del15*** – Number of flights delayed ($\geq$ 15minutes late)
  *constraints*: None
- ***carrier_ct*** – Number of flights delayed due to air carrier
  *constraints*: None
- ***weather_ct*** – Number of flights delayed due to weather
  *constraints*: None
- ***nas_ct*** – Number of flights delayed due to National Aviation System
  *constraints*: None
- ***security_ct*** – Number of flights delayed due to security
  *constraints*: None
- ***late_aircraft_ct*** – Number of flights delayed due to a previous flight using the same aircraft being late.
  *constraints*: None
- ***arr_cancelled*** – Number of cancelled flights.
  *constraints*: None
- ***arr_diverted*** – Number of diverted flights.
  *constraints*: None
- ***arr_delay*** – Total time (minutes) of delayed flights.
  *constraints*: None
- ***carrier_delay*** – Total time (minutes) of delayed flights due to air carrier.
  *constraints*: None
- ***weather_delay*** – Total time (minutes) of delayed flights due to weather.
  *constraints*: None
- ***nas_delay*** – Total time (minutes) of delayed flights due to National Aviation System.
  *constraints*: None
- ***security_delay*** – Total time (minutes) of delayed flights due to security.
  *constraints*: None
- ***late_aircraft_delay*** – Total time (minutes) of delayed flights due to a previous flight using the same aircraft being late.
  *constraints*: None
- ***year*** – Flight statistics of given year
  *constraints*: not null
- ***month*** – Flight statistic of given month
  *constraints*: not null
- ***airline_id*** – ID to identify airline
  *constraints*: foreign key references airline(id) on delete cascade on update cascade
- ***airport_id*** – ID to identify airport
  *constraints*: foreign key references airport(id) on delete cascade on update cascade

Below is the ER diagram for the database, which shows relations and its attributes as well as the relationships.

## IV. BCNF Provement

Let us consider a relation $R(X, Y, Z)$ where $X$, $Y$, $Z$ are the attributes of the relation. The table is said to be in BCNF if it satisfies below two conditions (1) It must satisfy 3NF (2) If $X \to Y$ and if X is not a super key, then it is said to violate BCNF.

Our database consists of five tables, and they are City, State, Airport, Airline and Flight. Let us analyze and validate our tables based on BCNF. If the left-hand side of the functional dependency is not a super key, then we should decompose the table.

**State Relation:** Here both the attributes of the tables are primary key, hence they are going to be super keys. So, this relation is BCNF compliant.

**City Relation:** Here $id$ and $city\_name$ are super keys and $state\_id$ is not a super key. We need to check whether there exists any functional dependency which contains state

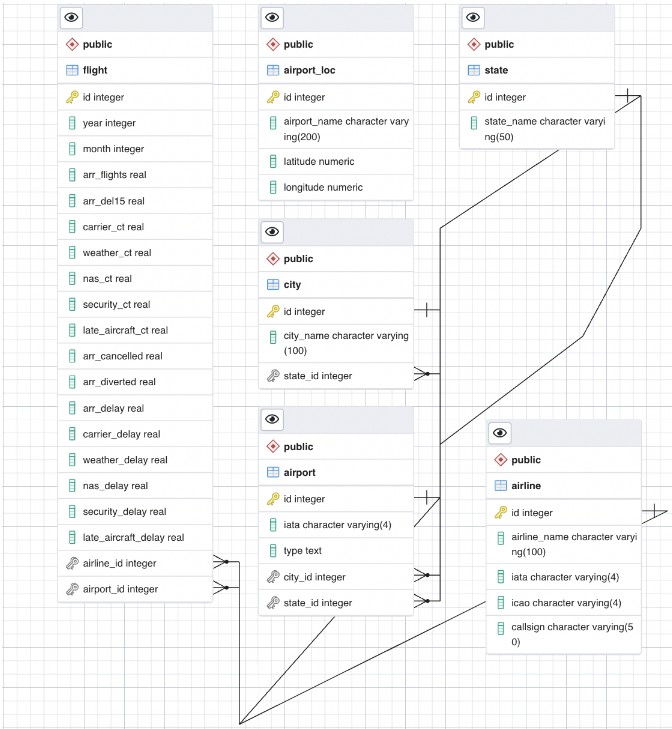


Fig. 1. ER Diagram

id on right hand side of dependency. The only functional dependency in this relation is $city\_name \rightarrow state\_id$. So, this relation is also BCNF compliant.

**Airport Relation:** Here $id$, $airport\_name$ are super keys and remaining all attributes are non-super keys. $iata$ is also a unique and not null attribute, so it is also a super key. Functional dependencies which we should consider is $longitude, latitude \rightarrow airport\_ame$. Since non super key attributes are present on left-hand side of this dependency, so we need to do BCNF decomposition. So we break Airport relation into two relations with $id$ as common attribute.

New set of relations are:

- airport relation: $(id, iata, type, city\_id, state\_id)$
- airport_loc relation: $(airport\_name, latitude, longitude)$

**Airline Relation:** Here $id$ is a super key and remaining all are non-super keys. But no two airlines can have same $iata$, $icao$, $callsign$, $airline\_name$ so then can also become super keys. So all dependencies in this relation contain super key in left side. Hence, this relation follows BCNF.

**Flight Relation:** In this relation only $id$ is a super key and remaining all attributes are non-super keys. There are no nontrivial dependencies so this relation follows BCNF.

## V. SQL Execution

To demonstrate data retrieval, we used 7 queries to get insights into the flights data including inserting, deleting and updating data.

- ***insert***

```
insert into flight (id, year,
    month, arr_flights, arr_del15,
    carrier_ct, weather_ct, nas_ct,
     security_ct, late_aircraft_ct,
     airline_id, airport_id)
values (
     (select max(id) from flight)
        +1, 2022, 2, 75, 3, 1,
        2,3,4,1,
     (select id from airline where
        airline_name like 'American
        ␣Airlines%'),
     (select id from airport_loc
        where airport_name like '
        John␣%␣Kennedy%'));
```

This query inserts a row in flight relation for a subset of attributes, where for `id` we use subquery to retrieve maximum value for the primary key and for `airport_id` and `airline_id` we use subquery to retrieve ID value for airline name 'American Airlines' and airport similar to 'John Kennedy'. Below is a result for insert verification (partial table shown).

| | id [PK] integer | year integer | month integer | arr_flights real |
|---|---|---|---|---|
| 1 | 80267 | 2022 | 2 | 75 |

Fig. 2. Insert Result

- ***delete***

```
delete from flight where
    airport_id=(select id from
    airport_loc where airport_name
    like 'George␣Bush%');
```

This query delete those entries from flight which has `airport_id` equals to that having 'George Bush' in the name, using subquery. Below is a screenshot for delete verification.

| | id [PK] integer | year integer | month integer | arr_flights real |
|---|---|---|---|---|
| | | | | |

Fig. 3. Delete Result

- ***update***

```
update airport set type='large'
    where id=(select id from
    airport_loc where airport_name
    like 'Lehigh␣Valley%');
```

This query update `type` attribute to 'large' in airport relation having `id` equals to `airport_name` starting with 'Lehigh Valley', which is done using subquery. We

| | id integer | iata character varying (4) | type text | city_id integer |
|---|---|---|---|---|
| 1 | 1 | ABE | large | 1 |

Fig. 4. Update Result

can verify the query result by displaying the subset of the matched row.

- ***select***

```
select airport_loc.airport_name,
    count(*) as total_flight from
    flight, airport_loc where
    flight.airport_id=airport_loc.
    id group by airport_loc.
    airport_name;
```

Using this select query, we find number of flights that flew from each airport in the past 4 years. We use `group_by` to aggregate rows with same `airport_id`. We project `airport_name` and total number of flights that flew from that airport in the result view. Query result is shown below.

| | airport_name character varying (200) | total_flight bigint |
|---|---|---|
| 1 | George Bush Intercontinental Houston Airport | 598 |
| 2 | Provo-Utah Lake International Airport | 49 |
| 3 | Evansville Regional Airport | 244 |
| 4 | Chippewa County International Airport | 49 |
| 5 | Owensboro Daviess County Airport | 47 |
| 6 | Bradley International Airport | 610 |
| 7 | Pierre Regional Airport | 34 |
| 8 | Rogue Valley International Medford Airport | 162 |
| 9 | Orlando International Airport | 467 |
| 10 | Easterwood Field | 111 |
| 11 | Columbus Metropolitan Airport | 65 |
| 12 | Fresno Yosemite International Airport | 289 |
| 13 | Lubbock Preston Smith International Airport | 238 |
| 14 | Will Rogers World Airport | 573 |

Fig. 5. Select Result 1

```
select s.state_name, count(*) as
    airports_num from (select *
    from airport natural join
    airport_loc) as a, state s
    where a.state_id=s.id group by
    s.state_name;
```

Using this select query, we find number of airports in each state in United States. We do natural join between `airport` and `airport_loc` using subquery and then perform theta join with the `state` on the condition that `id` and `state_id` matches and then it is grouped by `state_name`. Query result is shown below.

```
select sum(arr_flights) as
    arrival_flight, sum(arr_del15)
    as delayed_flight, sum(
```

| | state_name character varying (50) | airports_num bigint |
|---|---|---|
| 1 | Oklahoma | 4 |
| 2 | Colorado | 11 |
| 3 | North Carolina | 10 |
| 4 | Mississippi | 5 |
| 5 | Florida | 18 |
| 6 | Delaware | 1 |
| 7 | Vermont | 1 |
| 8 | Nevada | 3 |
| 9 | Louisiana | 7 |
| 10 | New York | 16 |
| 11 | West Virginia | 4 |
| 12 | New Jersey | 3 |
| 13 | South Carolina | 6 |
| 14 | Hawaii | 5 |

Fig. 6. Select Result 2

```
    weather_ct) as weather_delay,
    sum(nas_ct) as nas_delay, sum(
    security_ct) as security_delay
    from flight group by month,
    year order by year, month;
```

In the above given query, we find the total flights arrived on time, flights delayed by 15 minutes, flights delayed due to weather, flights delayed due to aviation system and flights delayed due to security. We calculate values for each month from January 2018 to January 2022, so total 48 months. We group results by `month` and `year` and then apply `sum` aggregate function. Then the results are ordered by `year` and `month` to show results starting from 2018 till 2022. Query result is shown below.

| | arrival_flight real | delayed_flight real | weather_delay real | nas_delay real | security_delay real |
|---|---|---|---|---|---|
| 1 | 567413 | 97043 | 4067.0293 | 30034.387 | 200.31003 |
| 2 | 518270 | 96755 | 3306.4185 | 32488.908 | 164.74995 |
| 3 | 608989 | 98116 | 2121.2893 | 30249.709 | 237.44998 |
| 4 | 592956 | NaN | 2470.0405 | 34419.695 | 162.01003 |
| 5 | 613322 | 117565 | 4367.5005 | 37307.805 | 205.41002 |
| 6 | 622454 | 133083 | 5771.84 | 40080.99 | 199.59998 |
| 7 | 641352 | NaN | 6308.137 | 42793.074 | 266.60007 |
| 8 | 641235 | NaN | 6240.579 | 46221.695 | 231.29997 |
| 9 | 583083 | 94502 | 3879.5686 | 33493.918 | 215.18 |
| 10 | 613361 | 101223 | 2546.3691 | 35825.637 | 128.39001 |
| 11 | 583223 | 111119 | 2981.429 | 38720.293 | 197.17001 |
| 12 | 590288 | 107702 | 2931.7988 | 34897.03 | 278.27002 |
| 13 | 580414 | NaN | 4107.5767 | 37282.727 | 182.52998 |
| 14 | 530028 | NaN | 4345.9277 | 40161.85 | 216.30003 |

Fig. 7. Select Result 3

```
with total_flights as (
```

```
        select sum(arr_flights+
            carrier_ct+weather_ct+
            nas_ct+security_ct+
            late_aircraft_ct) from
            flight
    );
    select sum(arr_flights)/(select *
        from total_flights)*100 as
        arr_flight_percent,
         sum(carrier_ct)/(select * from
             total_flights)*100 as
            weather_ct_percent,
         sum(nas_ct)/(select * from
            total_flights)*100 as
            nas_ct_percent,
         sum(security_ct)/(select *
            from total_flights)*100 as
            security_ct_percent,
         sum(late_aircraft_ct)/(select
            * from total_flights)*100
            as late_aircraft_ct_percent
    from flight;
```

In the above given query, we calculate the percentage of each attribute compared to total number of flights. We use `with as` clause to name the subquery block. We then use that named subquery in the main select query to divide total value of the attribute by it and multiply by 100 to get the percentage. The output screenshot is below

| | arr_flight_percent double precision | weather_ct_percent double precision | nas_ct_percent double precision | security_ct_percent double precision | late_aircraft_ct_percent double precision |
|---|---|---|---|---|---|
| 1 | 85.81363558769226 | 4.471002891659737 | 4.287615790963173 | 0.038334334385581315 | 4.855005815625191 |

Fig. 8. Select Result 4

- ***role*** Since we do not want any unauthorized update to the database, we create a new role admin, which is given to administrator and the automated programs that insert data into the database. While users of the database get only read privilege.

```
    create role admin;
    grant select, update, delete,
        truncate, insert on state, city
        , airline, airport, airport_loc
        , flight to admin;
```

## VI. Query Execution Analysis

We have a select query, where we try to find number of entries in the flight relation having `airport_name` start with 'Alex'.

```
select count(*) from flight f,
    airport_loc a where f.airport_id=a
    .id and a.airport_name like 'Alex%
    ';
```

| | QUERY PLAN text |
|---|---|
| 1 | Aggregate (cost=2095.98..2095.99 rows=1 width=8) (actual time=26.790..26.791 rows=1 loops=1) |
| 2 | [...] -> Hash Join (cost=8.68..2095.44 rows=215 width=0) (actual time=0.069..26.762 rows=151 loops=1) |
| 3 | [...] Hash Cond: (f.airport_id = a.id) |
| 4 | [...] -> Seq Scan on flight f (cost=0.00..1873.67 rows=80267 width=4) (actual time=0.006..9.083 rows=79669 loops=1) |
| 5 | [...] -> Hash (cost=8.66..8.66 rows=1 width=4) (actual time=0.053..0.054 rows=1 loops=1) |
| 6 | [...] Buckets: 1024 Batches: 1 Memory Usage: 9kB |
| 7 | [...] -> Seq Scan on airport_loc a (cost=0.00..8.66 rows=1 width=4) (actual time=0.007..0.050 rows=1 loops=1) |
| 8 | [...] Filter: ((airport_name)::text ~~ 'Alex%'::text) |
| 9 | [...] Rows Removed by Filter: 372 |
| 10 | Planning Time: 0.233 ms |
| 11 | Execution Time: 26.831 ms |

Fig. 9. Query Plan before Optimization

From the figure we can see that `Hash Join` is taking the longest time and overall execution time for this query is 26ms.

In the query PostgreSQL does random access on `airport_id`, but since there is no index, it takes time to find entries that matches the condition. So, we create index on `airport_id` on flight relation.

```
create index flight_index on flight (
    airport_id);
```

And now we analyze the query plan, we see significant execution time drop to 0.2ms. This is shown in Fig. 10.

| | QUERY PLAN text |
|---|---|
| 1 | Aggregate (cost=451.21..451.22 rows=1 width=8) (actual time=0.205..0.206 rows=1 loops=1) |
| 2 | [...] -> Nested Loop (cost=0.29..450.68 rows=214 width=0) (actual time=0.037..0.184 rows=151 loops=1) |
| 3 | [...] -> Seq Scan on airport_loc a (cost=0.00..8.66 rows=1 width=4) (actual time=0.010..0.065 rows=1 loops=1) |
| 4 | [...] Filter: ((airport_name)::text ~~ 'Alex%'::text) |
| 5 | [...] Rows Removed by Filter: 372 |
| 6 | [...] -> Index Only Scan using flight_index on flight f (cost=0.29..439.88 rows=214 width=4) (actual time=0.025..0.101 r... |
| 7 | [...] Index Cond: (airport_id = a.id) |
| 8 | [...] Heap Fetches: 47 |
| 9 | Planning Time: 1.131 ms |
| 10 | Execution Time: 0.236 ms |

Fig. 10. Query Plan after Optimization

## VII. Dashboard

To connect database to real world entity and perform operations, we created a web application using Python Flask that connects to PostgreSQL database. It is hosted on Heroku PaaS.

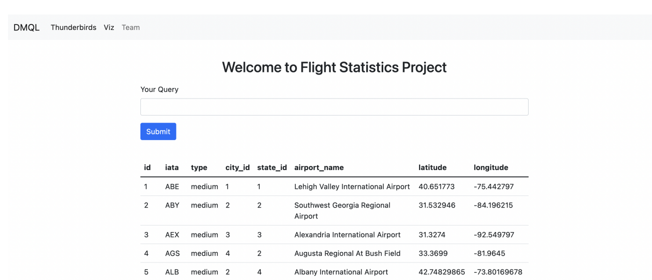


Fig. 11. Dashboard Homepage

On the homepage, we can run select queries and get the result in well formatted table. On the `/viz` page, we created visualization for select queries mentioned in section V. All the visualization are created in real-time.

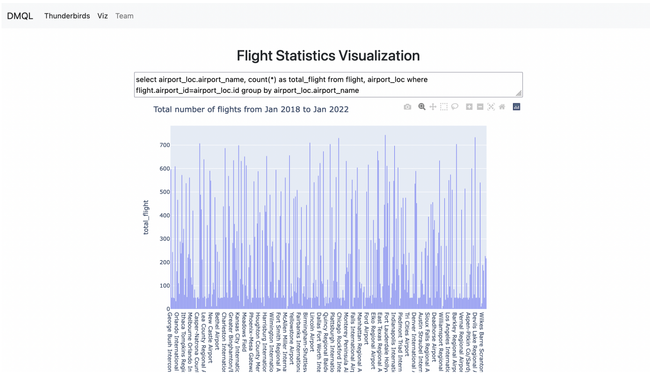


Fig. 12. Query Visualization

Dashboard can be reached at https://dmql-dashboard.herokuapp.com/. Since it is hosted on free plan, site takes few minutes to load on the first time.