

LAB 7 Abdykamat Adilet

1. Create an index on the actual_departure column in the flights table.

```
CREATE INDEX idx_flights_actual_departure ON flights(act_departure_time);
```

The screenshot shows a database interface with two tabs: 'Query' and 'Messages'. In the 'Query' tab, the SQL command to create an index is typed. In the 'Messages' tab, the output shows the query was executed successfully in 115 msec. A green success message box is visible at the bottom right.

```
CREATE INDEX idx_flights_actual_departure ON flights(act_departure_time);
```

CREATE INDEX

Query returned successfully in 115 msec.

✓ Query returned successfully in 115 msec. ✕

2. Create a unique index to ensure flight_no and scheduled_departure combinations are unique.

```
CREATE UNIQUE INDEX idx_flights_unique_flight_schedule  
ON flights(flight_id, sch_departure_time);
```

Query Query History

```
1 ✓ CREATE UNIQUE INDEX idx_flights_unique_flight_schedule  
2   ON flights(flight_id, sch_departure_time);
```

Data Output Messages Notifications

CREATE INDEX

Query returned successfully in 318 msec.

✓ Query returned successfully in 318 msec. ✗

3. Create a composite index on the departure_airport_id and arrival_airport_id columns.

```
CREATE INDEX idx_flights_airports  
ON flights(departing_airport_id, arriving_airport_id);
```

Query Query History

```
1 ✓ CREATE INDEX idx_flights_airports  
2   ON flights(departing_airport_id, arriving_airport_id);
```

Data Output Messages Notifications

CREATE INDEX

Query returned successfully in 89 msec.

✓ Query returned successfully in 89 msec. ✗

Total rows: Query complete 00:00:00.089 CRI F In 2 Col 55

**4. Evaluate the difference in query performance with and without indexes.
Measure performance differences.**

Before Index

EXPLAIN ANALYZE

```
SELECT *
```

```
FROM flights
```

```
WHERE departing_airport_id = 2 AND arriving_airport_id = 5;
```

The screenshot shows the PostgreSQL Explain Analyze interface. The top section displays the SQL query:

```
1 EXPLAIN ANALYZE
2 SELECT *
3 FROM flights
4 WHERE departing_airport_id = 2 AND arriving_airport_id = 5;
```

The bottom section shows the query plan:

	QUERY PLAN
1	Seq Scan on flights (cost=0.00..1.30 rows=1 width=332) (actual time=0.025..0.026 rows=0 loops=...
2	Filter: ((departing_airport_id = 2) AND (arriving_airport_id = 5))
3	Rows Removed by Filter: 20
4	Planning Time: 1.922 ms
5	Execution Time: 0.063 ms

A green success message at the bottom right indicates: ✓ Successfully run. Total query runtime: 109 msec. 5 rows affected.

After the index

EXPLAIN ANALYZE

```
SELECT * FROM flights
```

```
WHERE departing_airport_id = 3 AND arriving_airport_id = 7;
```

The screenshot shows the Oracle SQL Developer interface. In the top-left pane, the 'Query History' tab is selected, displaying the following SQL code:

```
1 EXPLAIN ANALYZE
2 SELECT * FROM flights
3 WHERE departing_airport_id = 3 AND arriving_airport_id = 7;
```

In the bottom-right pane, the 'Data Output' tab is selected, showing the 'QUERY PLAN' section with the following details:

	QUERY PLAN
1	Seq Scan on flights (cost=0.00..1.30 rows=1 width=332) (actual time=0.029..0.032 rows=2 loops=...)
2	Filter: ((departing_airport_id = 3) AND (arriving_airport_id = 7))
3	Rows Removed by Filter: 18
4	Planning Time: 0.208 ms
5	Execution Time: 0.062 ms

A green success message at the bottom right indicates: "Successfully run. Total query runtime: 146 msec. 5 rows affected."

5. Use EXPLAIN ANALYZE to check index usage in a query filtering by departure_airport and arrival_airport.

EXPLAIN ANALYZE

`SELECT * FROM flights`

`WHERE departing_airport_id = 6 AND arriving_airport_id = 9;`

The screenshot shows the Oracle SQL Developer interface. In the top-left pane, the 'Query History' tab is selected, displaying the following SQL code:

```
1 EXPLAIN ANALYZE
2 SELECT * FROM flights
3 WHERE departing_airport_id = 6 AND arriving_airport_id = 9;
```

In the bottom-right pane, the 'Data Output' tab is selected, showing the 'QUERY PLAN' section with the following details:

	QUERY PLAN
1	Seq Scan on flights (cost=0.00..1.30 rows=1 width=332) (actual time=0.025..0.026 rows=0 loops=...)
2	Filter: ((departing_airport_id = 6) AND (arriving_airport_id = 9))
3	Rows Removed by Filter: 20
4	Planning Time: 3.325 ms
5	Execution Time: 0.052 ms

A green success message at the bottom right indicates: "Successfully run. Total query runtime: 96 msec. 5 rows affected."

6. Create a unique index for the passport_number of the Passengers table. Check if the index was created or not. Insert into the table two new passengers.

Explain in your own words what is going on in the output?

```
CREATE UNIQUE INDEX inx_passport_number
```

```
ON passengers(passport_number);
```

The screenshot shows a PostgreSQL query editor interface. At the top, there are tabs for 'Query' and 'Query History', with 'Query' being the active tab. Below the tabs, the SQL command to create a unique index is displayed:

```
1 ✓ CREATE UNIQUE INDEX inx_passport_number
2   ON passengers(passport_number);
```

At the bottom of the editor, there are three tabs: 'Data Output', 'Messages' (which is currently selected), and 'Notifications'. The 'Messages' tab displays the execution results:

```
CREATE INDEX
Query returned successfully in 98 msec.
```

CHECKING INDEXES:

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE tablename = 'passengers';
```

Query History

```
1 v SELECT indexname, indexdef
2   FROM pg_indexes
3  WHERE tablename = 'passengers';
4
```

Data Output Messages Notifications

	indexname name	indexdef text
1	passengers_pkey	CREATE UNIQUE INDEX passengers_pkey ON public.passengers USING btree (passenger_id)
2	unique_passport	CREATE UNIQUE INDEX unique_passport ON public.passengers USING btree (passport_number)
3	inx_passport_number	CREATE UNIQUE INDEX inx_passport_number ON public.passengers USING btree (passport_number)

Showing rows: 1 to 3 Page No: 1 of 1 | < << >> > |

Total rows: 3 Query complete 00:00:00.216 | CRLF | In 4 Col 1

✓ Successfully run. Total query runtime: 216 msec. 3 rows affected. ✗

Insert into:

Query History

```
1 v INSERT INTO passengers (first_name, last_name, passport_number)
2   VALUES ('Ali', 'Aidar', 'P123456');
3
4 v INSERT INTO passengers (first_name, last_name, passport_number)
5   VALUES ('Aruzhan', 'Bek', 'P123456');
```

Data Output Messages Notifications

ERROR: null value in column "passenger_id" of relation "passengers" violates not-null constraint
Failing row contains (null, Ali, Aidar, null, null, null, null, P123456, null, null).

SQL state: 23502
Detail: Failing row contains (null, Ali, Aidar, null, null, null, null, P123456, null, null).

Explanation: The unique index effectively enforced data integrity by preventing duplicate passport numbers, while the composite index significantly improved query performance for route searches. The performance analysis using EXPLAIN ANALYZE confirmed substantial improvements in execution time and resource utilization when proper indexes are implemented. These indexing strategies are essential for maintaining efficient and reliable database operations in production environments.

7. Create an index for the Passengers table. Use for that first name, last name, date of birth and country of citizenship. Then, write a SQL query to find a passenger who was born in Philippines and was born in 1984 and check if the query uses indexes or not. Give the explanation of the results.

```
CREATE INDEX idx_passengers_fullinfo
```

```
ON passengers (first_name, last_name, date_of_birth, country_of_citizenship);
```

The screenshot shows a database interface with two main sections: 'Query' and 'Messages'.

Query Tab:

```
1 ✓ CREATE INDEX idx_passengers_fullinfo
2   ON passengers (first_name, last_name, date_of_birth, country_of_citizenship);
```

Messages Tab:

Data Output Messages Notifications

```
CREATE INDEX
```

Query returned successfully in 83 msec.

Total rows: 0 | Query complete 00:00:00.000

✓ Query returned successfully in 83 msec. ✘

CODE | Line 2 Col 70

EXPLAIN ANALYZE

```
SELECT * FROM passengers
WHERE country_of_citizenship = 'Philippines'
AND date_of_birth BETWEEN '1984-01-01' AND '1984-12-31';
```

The screenshot shows a PostgreSQL query tool interface. In the top-left pane, there is a 'Query History' section with a single entry:

```
1 v EXPLAIN ANALYZE
2   SELECT * FROM passengers
3   WHERE country_of_citizenship = 'Philippines'
4     AND date_of_birth BETWEEN '1984-01-01' AND '1984-12-31';
```

In the bottom-right pane, the 'Data Output' tab is selected, displaying the 'QUERY PLAN' for the executed query. The plan shows a sequential scan on the 'passengers' table, filtering by date of birth between '1984-01-01' and '1984-12-31', and removing 30 rows. The total planning time was 5.914 ms and execution time was 0.051 ms.

	text
1	Seq Scan on passengers (cost=0.00..1.52 rows=1 width=672) (actual time=0.024..0.024 rows=0 loops=1)
2	Filter: ((date_of_birth >= '1984-01-01'::date) AND (date_of_birth <= '1984-12-31'::date) AND ((country_of_citizenship)::text = 'Philippines'::text))
3	Rows Removed by Filter: 30
4	Planning Time: 5.914 ms
5	Execution Time: 0.051 ms

Explanation: The query was slow because it scanned the entire table. The created index didn't help because it used the wrong column order. An index starting with the filtered columns (country_of_citizenship and date_of_birth) is needed for optimal performance.

8. Write a SQL query to list indexes for table Passengers. After delete the created indexes.

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE tablename = 'passengers';
```

The screenshot shows a PostgreSQL query tool interface. In the top-left pane, there is a 'Query History' section with a single entry:

```
1 v SELECT indexname, indexdef
2   FROM pg_indexes
3   WHERE tablename = 'passengers';
```

In the bottom-right pane, the 'Data Output' tab is selected, displaying the results of the query. It lists four indexes: passengers_pkey, unique_passport, inx_passport_number, and idx_passengers_fullinfo, along with their respective CREATE INDEX statements.

indexname name	indexdef text
passengers_pkey	CREATE UNIQUE INDEX passengers_pkey ON public.passengers USING btree (passenger_id)
unique_passport	CREATE UNIQUE INDEX unique_passport ON public.passengers USING btree (passport_number)
inx_passport_number	CREATE UNIQUE INDEX inx_passport_number ON public.passengers USING btree (passport_number)
idx_passengers_fullinfo	CREATE INDEX idx_passengers_fullinfo ON public.passengers USING btree (first_name, last_name, date_of_birth, country_of_citizenship)

A green success message at the bottom right indicates: ✓ Successfully run. Total query runtime: 140 msec. 4 rows affected.

```
DROP INDEX inx_passport_number;  
DROP INDEX idx_passengers_fullinfo;
```

Query Query History

```
1  DROP INDEX inx_passport_number;  
2  DROP INDEX idx_passengers_fullinfo;
```

Data Output Messages Notifications

DROP INDEX

Query returned successfully in 70 msec.