Part 1: Theory Questions 1.

Database Design • Design a database schema to store data for European airports, flight schedules, flight statuses, and delays. • Describe the tables (e.g., Airports, Flights, Airlines, FlightStatus) and their relationships. • Explain how you would ensure data accuracy and maintain scalability. 2. Data Collection Strategy • How would you collect and store information on all European airports (including IATA code, ICAO code, country, etc.)? • Suggest methods for gathering real-time flight data. You may propose using ChatGPT API for scraping, ADB data, or third-party APIs like FlightAware. • Explain how you would handle missing, delayed, or inconsistent data. 3. Flight Monitoring and Claim Identification • Propose a system that monitors flights from European airports and flags delays of more than 2 hours. • Describe the technical approach for real-time monitoring, data updates, and alerts. • Suggest how the system can efficiently store and manage large volumes of data. 4. Future API Development • Describe how you would design and implement a scalable API to provide flight data across Europe on a daily basis. • Suggest how you would ensure API security, availability, and reliability.

Answers

Database Design

Schema Design (Optimized for Your Flight Data)

Airports Table

Stores airport-related details.

sql

CopyEdit

```
CREATE TABLE public.airports (
   airport_id SERIAL PRIMARY KEY,
   name TEXT NOT NULL,
   IATA_code VARCHAR(3) UNIQUE,
   ICAO_code VARCHAR(4) UNIQUE,
   country TEXT NOT NULL,
   city TEXT NOT NULL,
   latitude DOUBLE PRECISION,
   longitude DOUBLE PRECISION
);
```

Flights Table

Stores flight details with references to departure and arrival airports.

sql

CopyEdit

```
CREATE TABLE public.flights (
    flight_id SERIAL PRIMARY KEY,
    airline_id INT REFERENCES airlines(airline_id),
    departure_airport_id INT REFERENCES airports(airport_id),
    arrival_airport_id INT REFERENCES airports(airport_id),
    scheduled_departure TIMESTAMP,
    scheduled_arrival TIMESTAMP,
    actual_departure TIMESTAMP,
    actual_arrival TIMESTAMP,
    status TEXT CHECK (status IN ('On-Time', 'Delayed', 'Cancelled'))
);
```

Airlines Table

Stores airline information.

sql

CopyEdit

```
CREATE TABLE public.airlines (
    airline_id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    IATA_code VARCHAR(2) UNIQUE,
    ICAO_code VARCHAR(3) UNIQUE
);
```

Flight Status Table

Tracks flight status and delay reasons.

sql

CopyEdit

```
CREATE TABLE public.flight_status (
    status_id SERIAL PRIMARY KEY,
    flight_id INT REFERENCES flights(flight_id),
    status TEXT CHECK (status IN ('On-Time', 'Delayed', 'Cancelled')),
    delay_reason TEXT,
    delay_duration INTERVAL
```

);

Ensuring Data Accuracy and Scalability

- **Normalization:** Data is divided into separate tables to remove redundancy.
- **Indexes:** Used on **IATA_code, flight_id, and scheduled_departure** to improve query performance.
- **Partitioning:** Flights data can be partitioned by **date** for better efficiency.
- **Data Validation:** Triggers and constraints ensure valid entries (e.g., checking ICAO/IATA codes).
- **Replication:** Can be set up to ensure **high availability and fault tolerance** in PostgreSQL.

---

2. Data Collection Strategy

Collecting and Storing European Airport Data

- **Using APIs:**
  - **OpenSky Network API** – Real-time ADS-B data for flights.
  - **AviationStack API** – Provides flight schedules and live status.
- **Database Sources:**
  - **OpenFlights Database** – Contains airport and airline information.
  - **Eurocontrol Data** – Provides air traffic statistics.
- **Web Scraping (If Required):**
  - **Using BeautifulSoup & Selenium** to scrape flight schedules from airline websites.

Methods for Gathering Real-Time Flight Data

- **OpenSky Network API:** Fetches real-time ADS-B flight data.
- **AviationStack API:** Provides global flight tracking.
- **Direct Airline APIs:** Some airlines provide **flight status APIs** (Lufthansa, Ryanair).
- **Scheduled API Polling:** Fetches new flight data every **5 minutes**.

Handling Missing, Delayed, or Inconsistent Data

- **Default Values:** If real-time API fails, fallback to last known status.
- **Imputation:** Use **historical flight delays** for estimated arrival times.
- **Alerts:** Trigger notifications if **data is missing or conflicting**.
- **Regular Updates:** API polling and updating flights every **few minutes**.

---

3. Flight Monitoring & Delay Identification

Proposed Monitoring System

- **Real-time API Fetching:** Fetches and updates **flight statuses** periodically.
- **Database Triggers:** Automatically update **delays** and **status changes** in PostgreSQL.
- **Notification System:** Alerts if **a flight is delayed by more than 2 hours**.

Technical Approach

- **Kafka or RabbitMQ** → Handles real-time flight data ingestion.
- **WebSockets or Push Notifications** → Sends **instant flight delay alerts** to users.
- **Cloud Storage (AWS S3, Azure Blob)** → Stores logs of **historical flight data**.

Efficient Data Storage & Management

- **Data Partitioning:** Flight data stored **by date** for faster retrieval.
- **Time-Series Databases: InfluxDB** for analyzing **flight logs over time**.
- **Indexing:** Added on frequently used fields (flight_id, status) to speed up queries.

---

4. Future API Development

Scalable API Design

Your API should follow **REST principles** with well-defined endpoints.

Example REST API Endpoints:

Endpoint	Description
`GET /flights?date=YYYY-MM-DD`	Fetch all flights on a given date.
`GET /flights/{flight_id}`	Fetch details of a specific flight.
`GET /airports/{IATA_code}`	Fetch airport details by IATA code.
`POST /flights`	Add new flight details.
`PATCH /flights/{flight_id}`	Update flight status (delayed, on-time, etc.).

GraphQL Alternative

For flexible queries, a **GraphQL API** could allow:

- Querying specific **fields** of flights.
- Fetching **related data (airport + flight status)** in a **single request**.

Ensuring Security, Availability & Reliability

- **Rate Limiting:** Prevents API misuse by restricting the number of requests per user.
- **Authentication:** Uses **OAuth 2.0** or **API Keys** for **secure access control**.

- **Load Balancing:** API requests are distributed across **multiple servers**.
- **Auto-Scaling:** Uses **AWS Lambda** or **Kubernetes** to **scale API traffic dynamically**.