**Flight Management System Architecture**

This document provides a comprehensive overview of the flight management system architecture, explaining each component and the data flow between them. The system is designed to receive flight price information from external sources, process price changes, notify users of relevant price changes, and manage user alerts.

**System Components**

**1. External Data Sources**

* **Flights APIs**: Third-party services providing real-time flight price information from airlines and price comparison websites.
* **Data Format**: Structured data containing flight details and pricing information.

**2. Ingestion Layer**

* **API Gateway**: Simple entry point that calls external APIs and forwards flight data directly to the ingestion pipeline.
* **Ingestion Queue**: High-throughput message queue (RabbitMQ) that receives flight documents in batches of 10,000.
* **Processing Load**: Handles up to 40 million flight documents in a processing cycle with a target completion time of 30 seconds.

**3. Price Processing Layer**

* **Price Workers**: Worker services that consume flight documents from the Ingestion Queue.
* **Redis Cache**: In-memory database storing the most recent price for each flight. Used to determine if a price has changed.
* **Changed Price Queue**: Message queue that receives only flights with price changes (approximately 20% of total documents, or 8 million).
* **Key Functions**:
  + Compare incoming prices with cached prices
  + Update cache with new prices
  + Publish changed prices to the Changed Price Queue

**4. Alert Processing Layer**

* **Alert Workers**: Services that process price changes and identify which users should be notified.
* **DB Router**: Routes database queries to the appropriate database shard based on origin-destination pairs.
* **DB Shards**: Horizontally partitioned database containing user alerts and preferences.
* **Notification Queue**: Message queue that receives notifications to be sent to users.
* **Key Functions**:
  + Match price changes against user alerts
  + Filter alerts based on user preferences and thresholds
  + Generate notifications for qualifying price changes

**5. Database Layer SQL**

* **DB Router**: Directs queries to the appropriate database shard to ensure scalability.
* **DB Shards**: Horizontally partitioned databases that store:
  + User information
  + Alert configurations
* **Partitioning Strategy**: Data is partitioned by:
  + Date ranges (for flights)
  + Origin-destination pairs (for geographic distribution)

**6. Notification Layer**

* **Notification Service**: Processes notification requests and manages delivery preferences.
* **Push Service**: Handles the actual delivery of push notifications to mobile devices.
* **Key Functions**:
  + Format notifications according to device requirements
  + Handle delivery receipts and failures
  + Manage notification throttling and prioritization

**7. Client Applications**

* **Mobile App**: User-facing application that displays flight information and receives push notifications.
* **Key Functions**:
  + Allow users to set up and manage price alerts
  + Receive and display push notifications

**8. Alert Management API**

* **Load Balancer**: Distributes incoming API requests across multiple API servers.
* **API Alerts Management**: RESTful API service that handles CRUD operations for user alerts.
* **Key Functions**:
  + Create new price alerts
  + Read existing alerts
  + Update alert parameters
  + Delete unwanted alerts
  + Authenticate users

**Data Flow**

**1. Price Ingestion Flow**

1. External flight price APIs provide data through direct API calls
2. Flight data is received and immediately passed to the ingestion pipeline
3. Data is published to the Ingestion Queue in batches of 10,000 documents
4. Volume: 40 million documents per processing cycle
5. Target completion time: 60 seconds total for all documents

**2. Price Processing Flow**

1. Price Workers consume documents from the Ingestion Queue
2. For each flight, workers check the Redis Cache to determine if the price has changed
3. If the price has changed:
   * Update the Redis Cache with the new price
   * Publish the flight with price change details to the Changed Price Queue
4. Volume: Approximately 8 million documents (20% with price changes)

**3. Alert Processing Flow**

1. Alert Workers consume documents from the Changed Price Queue
2. Workers use the DB Router to query the appropriate database shard
3. The database query identifies users with matching alerts based on:
   * Origin and destination
   * Date range
   * Price thresholds
4. For each matching alert, a notification message is created
5. Notification messages are sent to the Notification Queue

**4. Notification Delivery Flow**

1. Notification Service consumes messages from the Notification Queue
2. Service formats notifications according to user preferences
3. Push Service delivers notifications to users' mobile devices
4. Delivery receipts are collected and processed
5. Target completion time: 60 seconds total for all notifications

**5. Alert Management Flow (CRUD)**

1. Users interact with the Mobile App to create or manage alerts
2. Mobile App sends requests to the API through the Load Balancer
3. API Alerts Management service processes the CRUD operations
4. Database updates are routed to the appropriate shard via the DB Router

**Performance Considerations**

**Processing Efficiency**

* Stage 1 (Price Ingestion): 40 million documents processed within 25-30 seconds
* Stage 2 (Price Processing): 40 million documents processed within 30 seconds
* Stage 3 (Alert Processing): 8 million documents processed within 10-30 seconds
* Stage 4 (Notification Delivery): Push notifications sent within 60 seconds
* Total end-to-end processing time: 125-150 seconds (well under the 5-minute requirement)
* Batch processing is used throughout:
  + 10,000 documents per batch to RabbitMQ
  + Batch Redis operations (MGET)
  + Batch SQL queries (approximately 500 documents per batch)

**Scaling Capabilities**

* Horizontal scaling: All components can be scaled by adding more instances
* Stateless design: Workers maintain no local state, allowing for easy scaling
* Message queues: Act as buffers during traffic spikes
* Database sharding: Distributes load across multiple database instances

**Redis Cache Optimization**

* In-memory storage for fast access (sub-millisecond)
* Stores all flight prices to ensure complete price comparison
* Cluster configuration for high availability and throughput

**Database Optimization**

* Table partitioning by date ranges for efficient queries
* Covering indexes for common query patterns
* Sharding by origin-destination pairs for distributable load

**Implementation Notes for Alert Management (Section 3)**

The Alert Management component should be implemented using:

1. ASP.NET Core Web API for the RESTful service
2. Entity Framework Core for database access
3. JWT-based authentication for security
4. Repository pattern for data access abstraction

**Key Models**

* User model with authentication details
* Alert model with flight criteria and notification preferences
* Notification model for tracking delivery status

**Essential Endpoints**

* POST /api/alerts - Create new alert
* GET /api/alerts - List user's alerts
* GET /api/alerts/{id} - Get specific alert
* PUT /api/alerts/{id} - Update alert
* DELETE /api/alerts/{id} - Delete alert