# Developing the Statistics Management System (SMS) Using the Database Server

The SMS system will process raw sensor data from the Baggage Handling System (BHS) and generate useful statistics, reports, and insights. Here's a step-by-step description of how to design and develop this system.

---

## 1. System Overview

**Inputs:**

- **Raw Sensor Data:** Conveyor system error counts, baggage counts, flow rates, flight details, etc.
- **Operator Input:** Commands for report generation, data queries, and system monitoring.

**Outputs:**

- **Statistical Reports:** Daily, weekly, monthly, and annual reports (e.g., baggage flow, system errors).
- **Real-Time Monitoring:** Dashboard for operational status and fault information.
- **Database Queries:** On-demand retrieval of data for specific airlines, flights, or time periods.

**Components:**

1. **Sensors and Devices:** Collect raw data from the BHS.
2. **Data Ingestion Layer:** Interface for receiving, validating, and preprocessing data.
3. **Database Server:** Core component for storing and processing data.
4. **Application Layer:** Software for processing data and generating reports.
5. **User Interface (UI):** Dashboard and reporting tools for operators.

---

## 2. Development Steps

### Step 1: Raw Sensor Data Collection

- **Data Sources:**
  - Conveyor system sensors for error counts and baggage flow.
  - Automated Bag Tag Readers (ATRs) for baggage details.
  - Flight data from BHS systems.
- **Data Format:**
  - Sensors typically output data as numerical values, logs, or real-time streams.
  - ATR data may be encoded in XML, JSON, or binary.
- **Data Collection Tools:**

- Use communication protocols such as **Modbus**, **OPC UA**, or **MQTT** to connect sensors to the system.
- A **data acquisition module** (middleware) will aggregate and normalize sensor data into a unified format.

**Step 2: Data Ingestion Layer**

- **Purpose:** Accept and validate raw data before storing it in the database.
- **Components:**
  - **Data Validation Module:** Checks for missing or invalid data.
  - **Preprocessing Module:** Converts raw data into structured formats.
  - **Queue System:** Implements a queuing service like **Apache Kafka** or **RabbitMQ** to handle high-throughput data streams.
- **Example Workflow:**
  1. Conveyor sensor sends baggage count: `{"conveyor_id": 1, "baggage_count": 15, "timestamp": "2025-01-05T14:00:00"}`.
  2. Data validation ensures values are within expected ranges.
  3. Preprocessing formats the data into SQL-compatible records.

**Step 3: Database Design**

- **Schema Design:** Create database tables to store structured data. Example tables:
  - **Baggage Statistics Table:**
    - `baggage_id`, `conveyor_id`, `flight_id`, `baggage_type`, `count`, `timestamp`.
  - **Error Log Table:**
    - `error_id`, `conveyor_id`, `error_type`, `timestamp`.
  - **Flight Information Table:**
    - `flight_id`, `airline`, `departure_time`, `arrival_time`, `baggage_count`.
- **Storage Type:**
  - **Transactional Data:** Stored in relational databases like PostgreSQL or MySQL for structured queries.
  - **Large Data Volumes:** Use **NoSQL databases** like MongoDB for unstructured or semi-structured data.
- **Indexing:**
  - Index fields frequently queried (e.g., `flight_id`, `timestamp`) to optimize retrieval times.

---

**Step 4: Data Processing and Analysis**

- **Real-Time Processing:** Use in-memory processing tools (e.g., **Apache Spark Streaming**, **Redis**) for immediate insights.
- **Batch Processing:** Schedule daily jobs to summarize data into hourly, daily, or weekly reports.

- **Statistical Analysis:**
    - Use **Python** libraries like Pandas and NumPy for calculations.
    - Example metrics: baggage flow rates, ATR reading rates.

---

### Step 5: Application Layer

- **Middleware Services:**
    - Develop REST APIs using **Django REST Framework** or **Flask** to provide access to database queries.
    - Example API endpoints:
        - `/get_baggage_statistics`: Returns baggage counts by flight.
        - `/get_error_logs`: Retrieves conveyor error logs.
- **Report Generation:**
    - Use **Python** libraries like **Matplotlib** or **ReportLab** for generating visual and PDF reports.
    - Automate daily report generation using cron jobs or task schedulers like **Celery**.

---

### Step 6: User Interface (UI)

- **Dashboard Design:**
    - Implement using **React**, **Angular**, or **Vue.js**.
    - Real-time charts for baggage flow, error counts, and flight statistics using libraries like **Chart.js** or **D3.js**.
- **Key Features:**
    - Filters for date range, flight ID, or airline.
    - Interactive graphs and downloadable reports.

---

### Step 7: Integration

- **Integration with BHS:**
    - Ensure secure communication between sensors and the database.
    - Implement data synchronization protocols for fault-tolerant data transfer.
- **Integration with SCT:**
    - Provide access to the database via APIs for querying and report retrieval.
    - Develop user authentication for secure access.

---

### Step 8: Testing and Deployment

- **Testing:**
  - Validate data ingestion with dummy sensor data.
  - Stress test the database with large volumes of simulated data.
  - Test API endpoints and UI for usability and responsiveness.
- **Deployment:**
  - Use containerization tools like **Docker** for deployment.
  - Set up load balancers (e.g., **Nginx**) to distribute traffic.

---

## 3. Workflow

1. **Data Collection:** Sensors send real-time baggage flow data.
   - Example: `{"conveyor_id": 101, "baggage_count": 50, "timestamp": "2025-01-05T14:30:00"}`.
2. **Data Ingestion:** Data is validated, preprocessed, and inserted into the database.
3. **Storage:** The database stores the data for future queries.
4. **Processing:** Python scripts generate hourly and daily summaries.
5. **Visualization:** The UI displays baggage flow rates and error counts in real time.
6. **Reporting:** Operators request a monthly baggage flow report, which is generated automatically by the system.

## Hardware selection

### Data base server (intel Xeon prefer )

Need 2 servers for Active – Passive architecture for redundancy. One server is active while other observe the active and update necessary fields for take over when faliour. They keep shared memory for data base and separate internal memory for application.

### Memory (need ssd)

Need shared memory for both servers. In this need fast communication with the servers because the latency may increase this architecture. But synchronized 2 separate memory for the server is not suitable because even though it is fast the synchronization is hard. Prefer Network Shard Memory for better redundancy. Need the capacity for backups and the required capacity of duration.

### Power supply

Need guaranteed power supply for the servers even for power failiour to ensure 24 *7 functionality.

### Sensor OS

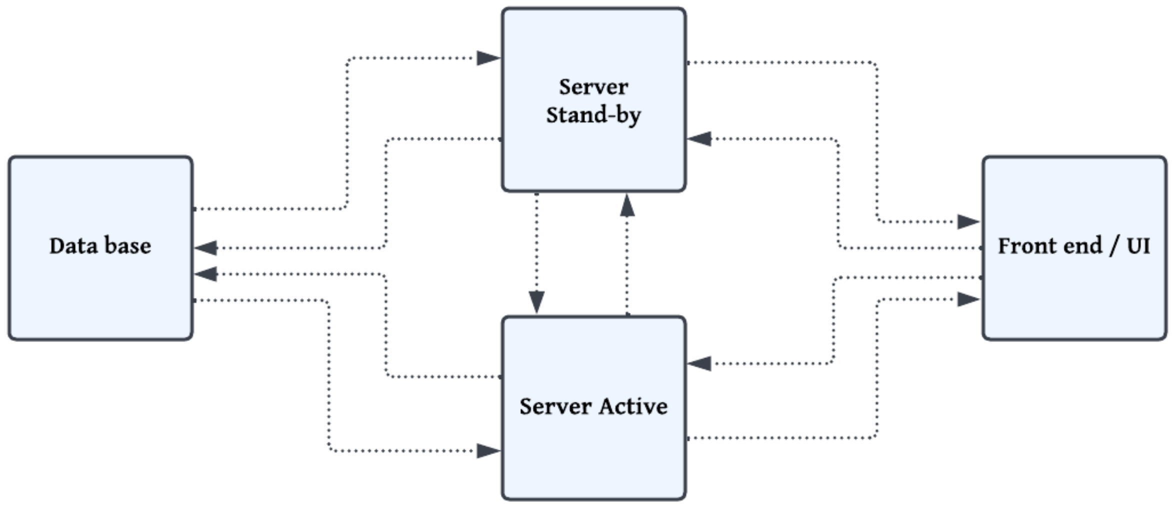
Ubuntu server prefer. Or Microsoft also fine.

### Monitor (2 separate monitors better)

For user interface and database management.

## Software for Front end, Back end and Database development

### Architecture



### Front end development

User interface for visualize the database and update forums and report generation request forums.

### Back end development

Extract required data for visualization from database, extract the sensor data and external system data and update the data base, process required data for report generation, process manual data update in the data base, do the sorting as required, provide interfaces for required external systems, develop backup system, develop redundant system for

### Database development

Develop related table structure, configure with back end for data editing, report generation, data manual update process, interface for external system, backup support.

## Hardware Selection Proposal

### 1. Database Servers

- **Specifications**
  - **Processor**: Intel Xeon E-2236 (6 cores, 3.4 GHz) or similar.
    - Suitable for moderate workloads with room for redundancy tasks.
  - **Memory (RAM)**: 32 GB DDR4 ECC per server.
    - Sufficient for database operations and light application handling.
  - **Storage**
    - **Primary (OS and Application)**: 256 GB NVMe SSD.
    - **Database (Shared Memory)**: 1 TB SSD (local or shared).
      - Use high-speed SSDs for better performance.
  - **Networking**
    - Dual 1 GbE ports for server-to-server and storage communication.
- **Architecture**
  - Two servers in an **Active-Passive mode** for redundancy.
  - Shared database storage is handled externally via NAS/SAN.

---

### 2. Shared Memory (Storage)

- **Storage Type**
  - **NAS/SAN Solution**:
    - Entry-level NAS: Synology DS920+ or QNAP TS-451D2.
    - Storage Capacity: 2 TB SSD in RAID 1 (mirrored) for redundancy.
  - **Connection**: 1 GbE with sufficient bandwidth for database access.

---

### 3. Power Supply

- **Specifications**
  - **UPS (Uninterruptible Power Supply)**
    - APC Smart-UPS SMT1500C or equivalent.
    - Online UPS with a minimum 30-minute runtime for two servers and storage.

---

### 4. Operating System

- **Preference**:
  - **Ubuntu Server LTS**: Lightweight and optimized for server environments.

    - **Alternative**: Windows Server Standard edition for GUI-based management, if needed.

---

## 5. Monitors

- **Specifications**:
    - Single 22-inch Full HD monitor (1920x1080) for occasional UI and database management.
    - Use KVM switches to share the monitor between the two servers if necessary.

---

## 6. Backup System

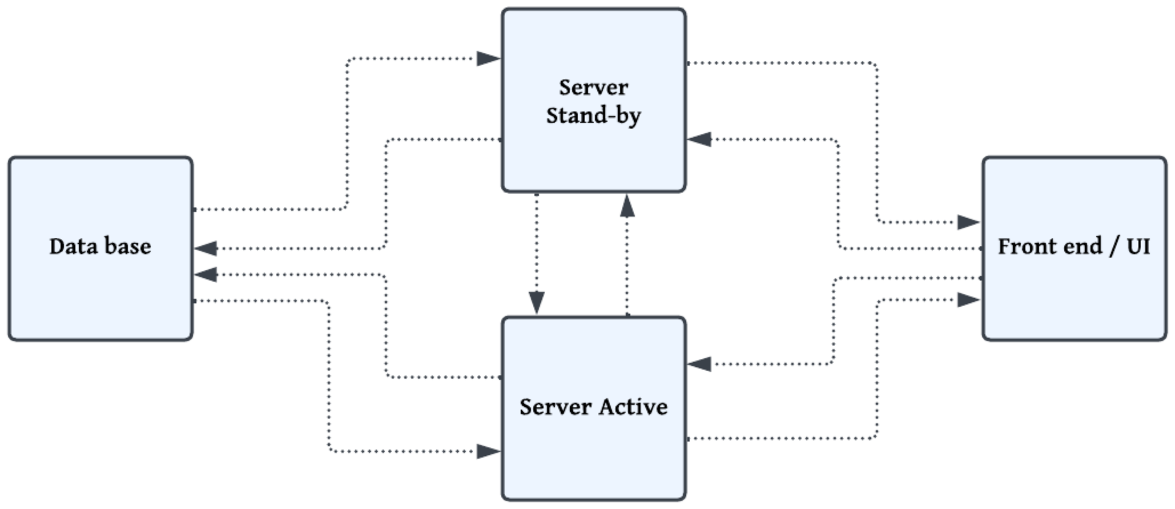
- **Hardware**
    - External USB HDD for local backups: 2 x 2 TB HDDs in a rotation schedule.
    - NAS/SAN solution to double as a backup storage device.

---

## 7. Network Infrastructure (optional)

- **Switch**:
    - Entry-level 8-port Gigabit switch (e.g., TP-Link TL-SG108 or similar).
- **Cabling**:
    - Use CAT6 Ethernet cables for consistent performance.

## Architecture



## 1. Front-End Development

### A. Responsibilities

1. **Dashboard Design**
   - Real-time monitoring of Baggage Handling System (BHS) status, including flight schedules, baggage flow rates, and error reports.
   - Dynamic visualizations for system operation data.
2. **User Interfaces**
   - Create interfaces for inputting and editing flight schedules.
   - Develop forms for generating and exporting statistical reports.
3. **Operational Data Display**
   - Display flight-wise baggage information dynamically.
   - Show alerts and notifications for system faults or emergencies.
4. **Responsive Design**
   - Ensure the interface is optimized for use on multiple device types (desktop, tablets).
5. **User Roles and Permissions**
   - Create login and role-based access control for operators and administrators.

### B. Tools and Technologies

- **Frameworks**: React or Angular
- **Styling**: CSS, Bootstrap, or Tailwind for responsive design.
- **APIs**: Integration with back-end APIs for dynamic data updates.

## 2. Back-End Development

The back-end will handle the data processing logic, system integration, and communication with external systems.

### A. Responsibilities

1. **Core Functionality**
   - Implement logic for flight schedule management, baggage tracking, and statistical analysis.
   - Develop modules for error handling and logging.
2. **System Integration**
   - Interface with external systems (FIS, DCS, CUPPS, Security Systems, BMS, MCS, etc.)
   - Ensure data exchange with these systems using protocols such as REST, SOAP, or specific middleware.
3. **Real-Time Data Processing**
   - Handle real-time data streams for baggage flow monitoring and system health status updates.
4. **API Development**:
   - Create RESTful APIs to serve data to the front end and external systems.
   - Ensure APIs are secure and optimized for high performance.
5. **Redundancy and Failover**
   - Implement server-side redundancy to ensure 24/7 availability..

### B. Tools and Technologies

- **Programming Languages**: Python (Flask/Django), Java (Spring Boot).

---

## 3. Database Development

The database will store all operational, statistical, and system-related data locally.

### A. Responsibilities

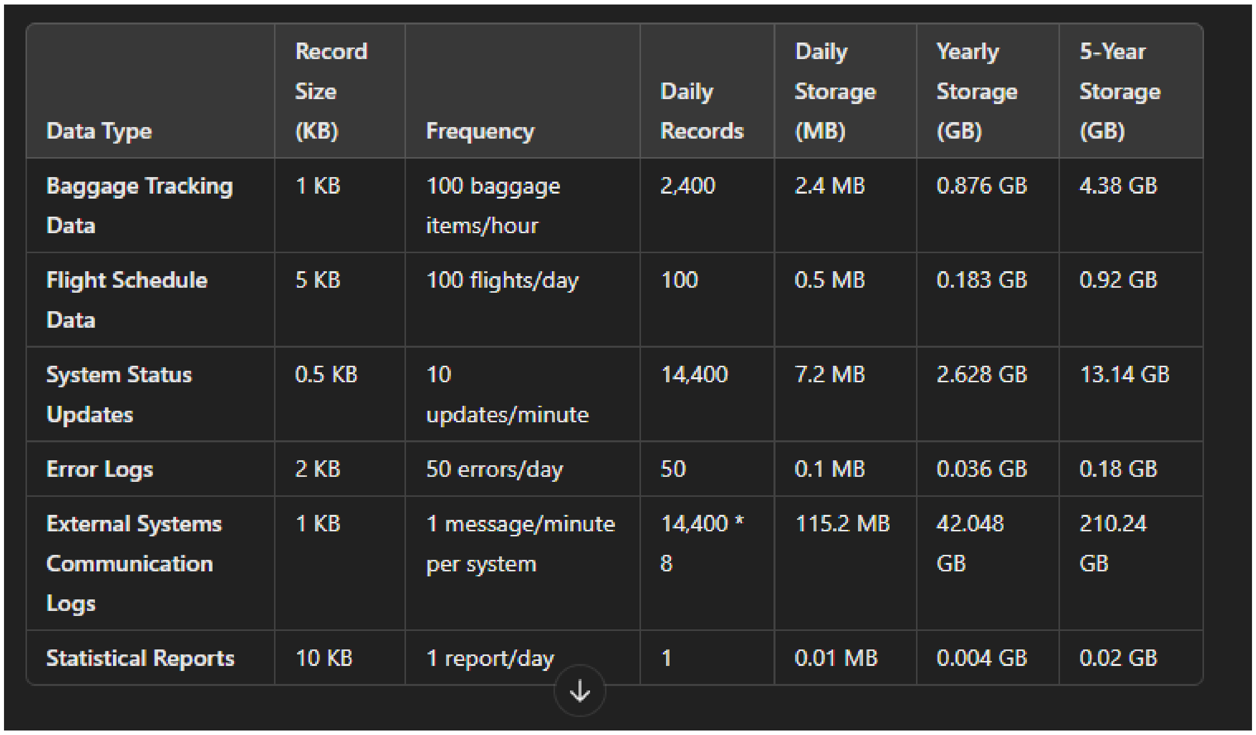
1. **Table Design**
   - Design tables for flight schedules, baggage data, error logs, system status, and report generation.
   - Ensure normalization to avoid redundancy.
2. **Data Storage**
   - Store real-time operational data, including baggage tracking and flight schedules.
   - Maintain historical data for generating statistical reports.

3. **Backup and Recovery**
   - Implement regular automated backups with redundancy to prevent data loss.
4. **Performance Optimization**
   - Optimize queries for real-time data access and reporting.
5. **Security and Compliance**
   - Implement role-based access control for database operations.

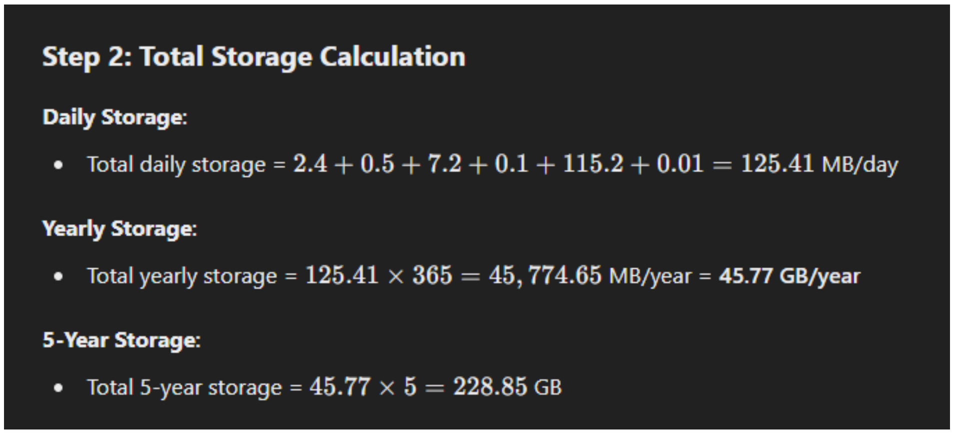
**B. Tools and Technologies**

- **Database Management System**: PostgreSQL, MySQL, or Microsoft SQL Server.
- **Backup Tools**: pg_dump (PostgreSQL), mysqldump (MySQL), or equivalent tools for automated backups.
- **Monitoring Tools**: pgAdmin, phpMyAdmin, or custom monitoring scripts.

## Storage calculation



| Data Type | Record Size (KB) | Frequency | Daily Records | Daily Storage (MB) | Yearly Storage (GB) | 5-Year Storage (GB) |
|---|---|---|---|---|---|---|
| Baggage Tracking Data | 1 KB | 100 baggage items/hour | 2,400 | 2.4 MB | 0.876 GB | 4.38 GB |
| Flight Schedule Data | 5 KB | 100 flights/day | 100 | 0.5 MB | 0.183 GB | 0.92 GB |
| System Status Updates | 0.5 KB | 10 updates/minute | 14,400 | 7.2 MB | 2.628 GB | 13.14 GB |
| Error Logs | 2 KB | 50 errors/day | 50 | 0.1 MB | 0.036 GB | 0.18 GB |
| External Systems Communication Logs | 1 KB | 1 message/minute per system | 14,400 * 8 | 115.2 MB | 42.048 GB | 210.24 GB |
| Statistical Reports | 10 KB | 1 report/day | 1 | 0.01 MB | 0.004 GB | 0.02 GB |



### Step 2: Total Storage Calculation

**Daily Storage:**

- Total daily storage = $2.4 + 0.5 + 7.2 + 0.1 + 115.2 + 0.01 = 125.41$ MB/day

**Yearly Storage:**

- Total yearly storage = $125.41 \times 365 = 45,774.65$ MB/year = **45.77 GB/year**

**5-Year Storage:**

- Total 5-year storage = $45.77 \times 5 = 228.85$ GB