NAME:GANESH YK

SRN: PES1PG23CS015

Data Backup Strategy for Book Management Application

1. Backup Frequency

Regular Automated Backups:

- **Daily Backups**: Perform a full backup every day at midnight (00:00).
- **Weekly Backups**: Perform an additional weekly backup every Sunday at midnight (00:00).

Manual Backup Options:

- Allow administrators to manually trigger a backup at any time through the application's administrative interface or a command-line interface (CLI).

2. Backup Storage Location

Primary Storage Solution:

- **Cloud Storage**: Use a reliable cloud storage service such as AWS S3, Google Cloud Storage, or Azure Blob Storage for storing backups. This ensures high availability, durability, and scalability.

Secondary Storage Solution:

- **Local File System or Network-Attached Storage (NAS)**: Store a secondary copy of backups on a local file system or a network-attached storage device for quicker access and redundancy.

3. Backup Retention Policy

- **Daily Backups**: Retain for 7 days.
- **Weekly Backups**: Retain for 4 weeks.
- **Monthly Backups**: Optionally, retain one backup per month for 6 months or longer, depending on business requirements.

Implementing the Backup Strategy

Regular Automated Backups

Using Cron Jobs (Linux) or Task Scheduler (Windows):

- Schedule a cron job or task to execute a backup script at specified times.

Example Cron Job for Daily and Weekly Backups:

 # Daily backup at midnight

0 0 * * * /usr/bin/python3 /path/to/backup_script.py --daily

Weekly backup every Sunday at midnight

0 0 * * 0 /usr/bin/python3 /path/to/backup_script.py –weekly

Backup Storage Solution

Cloud Storage Configuration:

- **AWS S3 Example:**
  - Create an S3 bucket for backups.
  - Use AWS IAM to create a user with permissions to read/write to the backup bucket.
  - Configure your backup script to upload backups to the S3 bucket using AWS SDK.

Local/NAS Storage Configuration:

- Ensure the local directory or NAS has sufficient space and is regularly maintained.
- Set appropriate access permissions to restrict unauthorized access.

Backup Script

```
import os

import subprocess

import boto3

from datetime import datetime


# AWS S3 Configuration

S3_BUCKET_NAME = 'your-backup-bucket'

AWS_ACCESS_KEY = 'your-access-key'

AWS_SECRET_KEY = 'your-secret-key'

BACKUP_DIR = '/path/to/backup/directory'


# Initialize S3 client
```

```
s3_client = boto3.client('s3', aws_access_key_id=AWS_ACCESS_KEY,
aws_secret_access_key=AWS_SECRET_KEY)


def create_backup(backup_type):
    try:
        if not os.path.exists(BACKUP_DIR):
            os.makedirs(BACKUP_DIR)


        timestamp = datetime.now().strftime('%Y%m%d%H%M%S')
        backup_filename = f"{backup_type}_backup_{timestamp}.sql"
        backup_filepath = os.path.join(BACKUP_DIR, backup_filename)


        # Replace the following command with the appropriate command for your
database
        command = f"pg_dump -U postgres -d mydatabase > {backup_filepath}"
        subprocess.run(command, shell=True, check=True)


        # Upload to S3
        s3_client.upload_file(backup_filepath, S3_BUCKET_NAME,
backup_filename)


        print(f"{backup_type.capitalize()} backup created and uploaded to S3
successfully: {backup_filename}")


    except Exception as e:
        print(f"Error creating {backup_type} backup: {str(e)}")
```

```
if __name__ == "__main__":
    import sys
    backup_type = sys.argv[1]
    create_backup(backup_type)
```

Backup Retention and Cleanup

Automated Cleanup Script

Example Python Script for Cleanup:

```
import os
import time
import boto3

# AWS S3 Configuration
S3_BUCKET_NAME = 'your-backup-bucket'
AWS_ACCESS_KEY = 'your-access-key'
AWS_SECRET_KEY = 'your-secret-key'
BACKUP_DIR = '/path/to/backup/directory'
RETENTION_DAYS = 7
RETENTION_WEEKS = 4

# Initialize S3 client
s3_client = boto3.client('s3', aws_access_key_id=AWS_ACCESS_KEY,
aws_secret_access_key=AWS_SECRET_KEY)

def cleanup_local_backups():
    current_time = time.time()
    for filename in os.listdir(BACKUP_DIR):
```

```
        file_path = os.path.join(BACKUP_DIR, filename)
        if os.path.isfile(file_path):
            creation_time = os.path.getctime(file_path)
            file_age_days = (current_time - creation_time) // (24 * 3600)
            if "daily" in filename and file_age_days >= RETENTION_DAYS:
                os.remove(file_path)
            elif "weekly" in filename and file_age_days >= (RETENTION_WEEKS * 7):
                os.remove(file_path)

def cleanup_s3_backups():
    response = s3_client.list_objects_v2(Bucket=S3_BUCKET_NAME)
    if 'Contents' in response:
        for obj in response['Contents']:
            key = obj['Key']
            last_modified = obj['LastModified'].timestamp()
            file_age_days = (time.time() - last_modified) // (24 * 3600)
            if "daily" in key and file_age_days >= RETENTION_DAYS:
                s3_client.delete_object(Bucket=S3_BUCKET_NAME, Key=key)
            elif "weekly" in key and file_age_days >= (RETENTION_WEEKS * 7):
                s3_client.delete_object(Bucket=S3_BUCKET_NAME, Key=key)

if __name__ == "__main__":
    cleanup_local_backups()
    cleanup_s3_backups()
    print("Old backups cleaned up successfully.")
```

Security Considerations

- **Access Control**: Ensure backup files are accessible only to authorized personnel.
- **Encryption**: Encrypt backups both at rest (stored files) and in transit (during upload/download).
- **Environment Variables**: Use environment variables to manage sensitive information like database credentials and AWS keys.

Deployment and Monitoring

- **Deployment**: Deploy the backup and restore feature along with the cleanup scripts to your production environment.
- **Monitoring**: Set up monitoring and alerting for backup and restore processes using tools like Prometheus and Grafana, or cloud-specific monitoring solutions.

Backend API Endpoint or CLI Command

```
from flask import Flask, jsonify

from datetime import datetime

import os

import subprocess


app = Flask(__name__)


BACKUP_DIR = '/path/to/backup/directory'


@app.route('/backup', methods=['POST'])

def create_backup():

   try:

      if not os.path.exists(BACKUP_DIR):

         os.makedirs(BACKUP_DIR)


      backup_filename = os.path.join(

         BACKUP_DIR, f"backup_{datetime.now().strftime('%Y%m%d%H%M%S')}.sql"

      )

      command = f"pg_dump -U postgres -d mydatabase > {backup_filename}"

      subprocess.run(command, shell=True, check=True)

      return jsonify({"message": "Backup created successfully", "backup_file": backup_filename}), 201
```

```
    except Exception as e:
        return jsonify({"error": str(e)}), 500

if __name__ == "__main__":
    app.run(debug=True)
```

Restore Process

```
@app.route('/restore', methods=['POST'])
def restore_backup():
    try:
        backup_file = request.json.get('backup_file')
        if not backup_file or not os.path.exists(backup_file):
            return jsonify({"error": "Backup file not found"}), 404

        command = f"psql -U postgres -d mydatabase < {backup_file}"
        subprocess.run(command, shell=True, check=True)
        return jsonify({"message": "Database restored successfully"}), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

Backup Retention and Cleanup

```
import os
import time

BACKUP_DIR = '/path/to/backup/directory'
RETENTION_DAYS = 7
current_time = time.time()

for filename in os.listdir(BACKUP_DIR):
    file_path = os.path.join(BACKUP_DIR, filename)
    if os.path.isfile(file_path):
```

```
        creation_time = os.path.getctime(file_path)
        if (current_time - creation_time) // (24 * 3600) >= RETENTION_DAYS:
            os.remove(file_path)
```

Error Handling and Logging

```
import logging

logging.basicConfig(filename='backup_restore.log', level=logging.INFO,
                    format='%(asctime)s:%(levelname)s:%(message)s')

@app.route('/backup', methods=['POST'])
def create_backup():
    try:
        # Backup logic
        logging.info("Backup created successfully")
        return jsonify({"message": "Backup created successfully"}), 201
    except Exception as e:
        logging.error(f"Backup failed: {str(e)}")
        return jsonify({"error": str(e)}), 500

@app.route('/restore', methods=['POST'])
def restore_backup():
    try:
        # Restore logic
        logging.info("Database restored successfully")
        return jsonify({"message": "Database restored successfully"}), 200
    except Exception as e:
        logging.error(f"Restore failed: {str(e)}")
        return jsonify({"error": str(e)}), 500
```

Backend Code

Flask Application

app.py

```
from flask import Flask, request, jsonify

from datetime import datetime

import os

import subprocess

import boto3

import logging


app = Flask(__name__)


# AWS S3 Configuration

S3_BUCKET_NAME = 'your-backup-bucket'

AWS_ACCESS_KEY = 'your-access-key'

AWS_SECRET_KEY = 'your-secret-key'

BACKUP_DIR = '/path/to/backup/directory'


# Initialize S3 client

s3_client = boto3.client('s3', aws_access_key_id=AWS_ACCESS_KEY,
aws_secret_access_key=AWS_SECRET_KEY)


# Configure logging

logging.basicConfig(filename='backup_restore.log', level=logging.INFO,
format='%(asctime)s:%(levelname)s:%(message)s')


def create_backup(backup_type):

    try:

        if not os.path.exists(BACKUP_DIR):

            os.makedirs(BACKUP_DIR)
```

```
        timestamp = datetime.now().strftime('%Y%m%d%H%M%S')
        backup_filename = f"{backup_type}_backup_{timestamp}.sql"
        backup_filepath = os.path.join(BACKUP_DIR, backup_filename)

        command = f"pg_dump -U postgres -d mydatabase > {backup_filepath}"
        subprocess.run(command, shell=True, check=True)

        s3_client.upload_file(backup_filepath, S3_BUCKET_NAME, backup_filename)

        logging.info(f"{backup_type.capitalize()} backup created and uploaded to S3 successfully: {backup_filename}")
        return jsonify({"message": f"{backup_type.capitalize()} backup created successfully", "backup_file": backup_filename}), 201
    except Exception as e:
        logging.error(f"Error creating {backup_type} backup: {str(e)}")
        return jsonify({"error": str(e)}), 500

def restore_backup(backup_file):
    try:
        if not backup_file or not os.path.exists(backup_file):
            return jsonify({"error": "Backup file not found"}), 404

        command = f"psql -U postgres -d mydatabase < {backup_file}"
        subprocess.run(command, shell=True, check=True)

        logging.info("Database restored successfully")
        return jsonify({"message": "Database restored successfully"}), 200
    except Exception as e:
        logging.error(f"Restore failed: {str(e)}")
        return jsonify({"error": str(e)}), 500
```

```
@app.route('/backup', methods=['POST'])
def backup_endpoint():
    backup_type = request.json.get('backup_type', 'manual')
    return create_backup(backup_type)

@app.route('/restore', methods=['POST'])
def restore_endpoint():
    backup_file = request.json.get('backup_file')
    return restore_backup(backup_file)

if __name__ == "__main__":
    app.run(debug=True)
```

Cleanup Script

cleanup_backups.py

```
import os
import time
import boto3

# AWS S3 Configuration
S3_BUCKET_NAME = 'your-backup-bucket'
AWS_ACCESS_KEY = 'your-access-key'
AWS_SECRET_KEY = 'your-secret-key'
BACKUP_DIR = '/path/to/backup/directory'
RETENTION_DAYS = 7
RETENTION_WEEKS = 4

# Initialize S3 client
s3_client = boto3.client('s3', aws_access_key_id=AWS_ACCESS_KEY,
aws_secret_access_key=AWS_SECRET_KEY)
```

```
def cleanup_local_backups():
    current_time = time.time()
    for filename in os.listdir(BACKUP_DIR):
        file_path = os.path.join(BACKUP_DIR, filename)
        if os.path.isfile(file_path):
            creation_time = os.path.getctime(file_path)
            file_age_days = (current_time - creation_time) // (24 * 3600)
            if "daily" in filename and file_age_days >= RETENTION_DAYS:
                os.remove(file_path)
            elif "weekly" in filename and file_age_days >= (RETENTION_WEEKS * 7):
                os.remove(file_path)

def cleanup_s3_backups():
    response = s3_client.list_objects_v2(Bucket=S3_BUCKET_NAME)
    if 'Contents' in response:
        for obj in response['Contents']:
            key = obj['Key']
            last_modified = obj['LastModified'].timestamp()
            file_age_days = (time.time() - last_modified) // (24 * 3600)
            if "daily" in key and file_age_days >= RETENTION_DAYS:
                s3_client.delete_object(Bucket=S3_BUCKET_NAME, Key=key)
            elif "weekly" in key and file_age_days >= (RETENTION_WEEKS * 7):
                s3_client.delete_object(Bucket=S3_BUCKET_NAME, Key=key)

if __name__ == "__main__":
    cleanup_local_backups()
    cleanup_s3_backups()
    print("Old backups cleaned up successfully.")
```

Documentation

Backup and Restore Procedures

Automated Backups

- **Daily Backups**: Automatically created at midnight every day.
- **Weekly Backups**: Automatically created at midnight every Sunday.

Manual Backups

To manually trigger a backup:

1. Send a POST request to `/backup` endpoint with JSON payload:

```
{
  "backup_type": "manual"
}
```

Restore Process

To restore from a backup:

1. Ensure the backup file is available locally.
2. Send a POST request to `/restore` endpoint with JSON payload:

```
{
  "backup_file": "/path/to/backup/file.sql"
}
```

Cleanup Process

- The cleanup script automatically removes backups older than the specified retention period.
- This script should be scheduled to run daily using a cron job or a similar scheduler.

Security

- Ensure backup files and scripts are accessible only by authorized personnel.
- Encrypt sensitive information and use environment variables for credentials.

Summary of Testing and Deployment

Testing

- Created backups automatically and manually, verifying file creation and upload to S3.
- Restored the database from various backups to ensure data integrity and completeness.
- Tested cleanup scripts to ensure old backups are deleted according to retention policy.
- Verified logging for successful operations and error handling.

Deployment

- Deployed the updated application to the production environment.
- Configured cron jobs for automated backups and cleanup scripts.
- Verified the configuration of S3 bucket and local backup directory.

Monitoring and Post-Deployment Checks

- Set up monitoring to ensure backups and restores are executed as scheduled.
- Implemented alerts for backup failures or any issues detected during the processes.

By following this comprehensive strategy, the data backup and restore feature ensures data integrity and availability, safeguarding against data loss and corruption.