Highlighting PostgreSQL: Robust Relational Database

PostgreSQL is a powerful, open-source object-relational database system known for its strong adherence to SQL standards, reliability, feature robustness, and performance. In your data platform, PostgreSQL serves multiple critical roles: as a reliable datastore for structured data, a repository for application-specific metadata (e.g., configurations), and most notably, as the **metastore for Apache Airflow**, storing DAG definitions, task states, and historical runs.

This guide will demonstrate basic and advanced use cases of PostgreSQL, leveraging your **Advanced Track** local environment setup and its integration with FastAPI, Airflow, and Spark.

Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook** and the **Progressive Path Setup Guide Deep-Dive Addendum**.

Basic Use Case: Application Metadata and Health Checks

Objective: To demonstrate PostgreSQL's role as a reliable storage for application-specific metadata and how basic connectivity can be verified.

Role in Platform: Provide a transactional and consistent store for structured data, configuration, and state management for various services.

Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure all Advanced Track services are running:** docker compose up --build -d from your project root. This includes the postgres service.
2. **Verify PostgreSQL is accessible:** Check Docker logs for the postgres container (docker compose logs postgres). It should show messages indicating readiness.
3. **Install psql (PostgreSQL interactive terminal):** If not already installed on your host, you can use docker exec to access it inside the container.
    - Install psql locally: [PostgreSQL Downloads]
    - Or execute inside container: docker exec -it postgres psql -U user -d main_db

Steps to Exercise:

1. **Connect to PostgreSQL:**
   ```
   psql -h localhost -p 5432 -U user -d main_db
   ```
   (Enter password when prompted).
   If connecting from inside another container via docker exec, use psql -h postgres -p 5432 -U user -d main_db as postgres is the service name in the Docker network.
2. **Verify Connection and Database Existence:**
    - Once connected, you should see the main_db=# prompt.

   - Run a simple query: SELECT current_database(); (Expected: main_db).
3. **Create a Sample Table for Application Metadata:**
   ```
   CREATE TABLE IF NOT EXISTS app_configs (
     config_key VARCHAR(255) PRIMARY KEY,
     config_value TEXT,
     last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP
   );
   ```
4. **Insert and Query Sample Metadata:**
   ```
   INSERT INTO app_configs (config_key, config_value) VALUES ('data_ingestion_enabled', 'true');
   INSERT INTO app_configs (config_key, config_value) VALUES ('max_batch_size', '1000');
   SELECT * FROM app_configs;
   ```

Verification:

- **psql Output:** You successfully connect, create a table, insert data, and retrieve it.
- **Docker Logs:** The postgres container logs show the executed queries and successful operations. This confirms basic connectivity and data manipulation capabilities.

Advanced Use Case 1: Apache Airflow Metastore Management

Objective: To demonstrate PostgreSQL's crucial role as the backend metastore for Apache Airflow, storing all DAG definitions, task instances, historical runs, and connections.
Role in Platform: Provide a robust, transactional, and scalable persistence layer for Airflow's operational data, enabling Airflow to function reliably and recover from failures.
Setup/Configuration:

1. **Ensure all Advanced Track services are running:** This includes postgres, airflow-init, airflow-webserver, airflow-scheduler, airflow-worker.
2. **Verify Airflow Initialization:** The airflow-init service should have completed successfully on startup (check its logs). This service typically runs airflow db upgrade and creates the admin user.
3. **Access Airflow UI:** http://localhost:8080 (login admin/admin).

Steps to Exercise:

1. **Access Airflow UI:** Navigate to http://localhost:8080.
2. **Observe DAGs and Runs:**
   - In the Airflow UI, go to "DAGs". You should see your defined DAGs (e.g., financial_ingestion_dag.py, insurance_transformation_dag.py).
   - Trigger a DAG (e.g., financial_ingestion_dag).
   - Observe the DAG's status and task instances in the UI.
3. **Inspect PostgreSQL for Airflow Metastore Data:**
   - Connect to the main_db in PostgreSQL: psql -h localhost -p 5432 -U user -d main_db.

  - **List Airflow tables:** \dt airflow.*; (Expected: A long list of tables prefixed with airflow_ or ab_, such as airflow_dag, airflow_taskinstance, airflow_log, etc.).
  - **Query DAG information:**
    SELECT dag_id, is_active FROM airflow_dag ORDER BY dag_id;

  - **Query Task Instance information:**
    SELECT dag_id, task_id, state, execution_date FROM airflow_taskinstance ORDER BY execution_date DESC LIMIT 10;

  - **Query Connections:** (Airflow stores connection details in the metastore).
    SELECT conn_id, conn_type, host, port FROM connection;

    Note: Sensitive details like passwords are encrypted in the metastore.
4. **Simulate Airflow Component Restart and Verify State Persistence:**
  - Stop airflow-webserver and airflow-scheduler: docker compose stop airflow-webserver airflow-scheduler.
  - Start them again: docker compose start airflow-webserver airflow-scheduler.
  - Access Airflow UI. The state of your DAGs (e.g., if a DAG was running, its state should be restored, or if a new run was scheduled, it should still be there) should be preserved.

Verification:

- **Airflow UI:** DAGs, task states, and connections are correctly displayed and managed, even after restarting Airflow components.
- **PostgreSQL Queries:** Direct queries to the main_db reveal the underlying Airflow metadata, confirming that PostgreSQL is robustly storing all Airflow operational data, which enables state persistence and recovery.

Advanced Use Case 2: Reference Data Management for Spark Jobs

Objective: To demonstrate how PostgreSQL can serve as a central repository for "reference data" (e.g., lookup tables, master data) that Spark batch or streaming jobs can join with to enrich raw incoming data.
Role in Platform: Provide a consistent and accessible source of static or slowly changing dimension data for enrichment processes within the data pipeline.
Setup/Configuration:

1. **Ensure PostgreSQL and Spark are running.**
2. **Populate Reference Data:** Connect to PostgreSQL and create a merchant_lookup table with sample data.
   Example SQL:
   ```
   CREATE TABLE IF NOT EXISTS merchant_lookup (
     merchant_id VARCHAR(50) PRIMARY KEY,
     merchant_name VARCHAR(255) NOT NULL,
   ```

```
    category VARCHAR(100)
);

INSERT INTO merchant_lookup (merchant_id, merchant_name, category) VALUES
('MER-ABC', 'Alpha Mart', 'Groceries'),
('MER-XYZ', 'Tech Gadgets Inc.', 'Electronics'),
('MER-123', 'HealthPlus Pharmacy', 'Healthcare');
```

3. **Spark Job:** Your pyspark_jobs/batch_transformations.py script (from "Highlighting Apache Spark" Advanced Use Case 1) already includes logic to connect to PostgreSQL and join with merchant_lookup.

Steps to Exercise:

1. **Ensure raw-data-bucket/financial_data_delta is populated** with some data that includes merchant_id.
2. **Submit the Spark batch transformation job:**
```
docker exec -it spark spark-submit \
  --packages io.delta:delta-core_2.12:2.4.0,org.postgresql:postgresql:42.6.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  /opt/bitnami/spark/jobs/batch_transformations.py \
  s3a://raw-data-bucket/financial_data_delta \
  s3a://curated-data-bucket/financial_data_curated_enriched
```
   (Note: Using a new output path to clearly show the enriched data).
3. **Monitor Spark Job:** Observe the console output for Spark logs indicating a successful read from PostgreSQL and the transformation process.
4. **Query Enriched Data:** After the Spark job completes, query the output Delta Lake table in MinIO using Spark SQL to confirm the enriched_category field is populated.
```
docker exec -it spark spark-sql \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  -e "SELECT transaction_id, merchant_id, enriched_category FROM
```

delta.\`s3a://curated-data-bucket/financial_data_curated_enriched\` WHERE merchant_id IS NOT NULL LIMIT 10;"

Verification:

- **Spark Job Logs:** Confirm that Spark successfully connected to PostgreSQL and performed the join operation.
- **Delta Lake Data:** The enriched_category column in the financial_data_curated_enriched table contains values from the merchant_lookup table, demonstrating successful data enrichment from PostgreSQL.

Advanced Use Case 3: Database Monitoring and Backup/Restore (Conceptual)

Objective: To conceptually demonstrate how PostgreSQL's operational health can be monitored and how backup and restore procedures are critical for disaster recovery, especially for the Airflow metastore.
Role in Platform: Ensure the availability and recoverability of critical structured data and Airflow's operational state, a key aspect of defining RPO/RTO.
Setup/Configuration (Conceptual Discussion):

1. **Monitoring:** Your grafana-alloy and grafana setup would typically include a Prometheus postgres_exporter (often run as a sidecar container to PostgreSQL) to collect PostgreSQL-specific metrics (e.g., active connections, query duration, replication lag, disk I/O).
   Example docker-compose.yml (conceptual for postgres_exporter):
   ```
   # services:
   #   postgres:
   #     # ... existing config
   #   postgres_exporter:
   #     image: prometheuscommunity/postgres-exporter:latest
   #     environment:
   #       DATA_SOURCE_NAME:
   "postgresql://user:password@postgres:5432/main_db?sslmode=disable"
   #     ports:
   #       - "9187:9187" # Exporter's metrics port
   #     depends_on:
   #       postgres:
   #         condition: service_healthy
   ```

   And in observability/alloy-config.river:
   ```
   # prometheus.scrape "postgres" {
   #   targets    = [{"__address__" = "postgres_exporter:9187"}]
   #   forward_to = [prometheus.remote_write.default.receiver]
   #   job        = "postgres_db"
   ```

}

2. **Backup/Restore:** PostgreSQL offers various backup strategies (e.g., pg_dump for logical backups, filesystem-level backups, streaming replication for continuous archiving and point-in-time recovery).

Steps to Exercise (Conceptual/Discussion):

1. **Discuss Database Monitoring in Grafana:**
   - Explain how metrics from postgres_exporter would be visualized in Grafana dashboards:
     - **Connection Usage:** See active connections, identify connection leaks.
     - **Query Performance:** Monitor long-running queries, identify slow queries.
     - **Replication Status:** (If replica set is configured) Monitor replication lag to ensure high availability.
     - **Disk I/O:** Track read/write operations to assess storage performance.
   - Relate these metrics to potential bottlenecks (e.g., high active connections could mean your FastAPI/Spark connection pool is too large or too small, leading to contention).
2. **Discuss Backup and Restore Procedure for Airflow Metastore:**
   - **Importance:** Emphasize that the Airflow metastore is critical. Its loss means losing all DAG run history, task states, and potentially DAG definitions.
   - **Backup Strategy:**
     - **Logical Backup (pg_dump):** Explain pg_dump -U user -d main_db > backup.sql to create a SQL dump.
     - **Physical Backup (Filesystem):** Discuss copying the entire data directory (/var/lib/postgresql/data) after stopping the DB or using pg_basebackup.
     - **Continuous Archiving:** For production, explain WAL (Write-Ahead Log) archiving for Point-In-Time Recovery (PITR).
   - **Restore Scenario (DR & Runbooks Deep-Dive Addendum, Section 6.3.1):**
     - Walk through the steps in the DR Runbook Example: Critical Database Restoration for PostgreSQL.
     - Highlight the importance of stopping applications (Airflow components), restoring the database, verifying integrity, and then restarting dependent applications in the correct order.
     - Emphasize the RPO (Recovery Point Objective) and RTO (Recovery Time Objective) implications for the Airflow metastore.

Verification (Conceptual):

- **Monitoring Insights:** Ability to articulate how PostgreSQL metrics would provide insights into database health and performance.
- **DR Understanding:** Clear explanation of PostgreSQL backup/restore mechanisms and their importance for the data platform's disaster recovery strategy, especially for the Airflow metastore, directly aligning with the DR & Runbooks addendum.

This concludes the guide for PostgreSQL.