Highlighting MinIO: Local S3-Compatible Object Storage

MinIO is an open-source object storage server that is compatible with Amazon S3 APIs. In your local data platform environment, MinIO serves as the crucial **simulated S3 data lake**, providing a cost-effective and convenient way to develop and test data pipelines that interact with object storage, including those leveraging Delta Lake. It offers the flexibility and scalability of object storage right on your development machine.
This guide will demonstrate basic and advanced use cases of MinIO, leveraging your **Advanced Track** local environment setup and its integration with Spark and other components.
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**, particularly MinIO's role as an S3 replacement.

Basic Use Case: Storing Raw Data from Spark Structured Streaming

Objective: To demonstrate how MinIO acts as the landing zone for raw data, receiving continuous appends from Spark Structured Streaming jobs, simulating data flowing into an S3 raw zone.
Role in Platform: Provide scalable, durable, and cost-effective storage for large volumes of raw, semi-structured, and structured data, accessible via S3 API.
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 minio and spark.
2. **Verify MinIO is accessible:** Navigate to http://localhost:9001 in your web browser. (Login with minioadmin/minioadmin).
3. **Ensure Spark Streaming jobs are running:** Your pyspark_jobs/streaming_consumer.py jobs (from the "Highlighting Apache Spark" document's Basic Use Case) should be actively consuming from Kafka and writing to s3a://raw-data-bucket/financial_data_delta and s3a://raw-data-bucket/insurance_data_delta.
4. **Generate activity:** Run python3 simulate_data.py to continuously send data to FastAPI, ensuring Kafka topics are populated for Spark to consume.

Steps to Exercise:

1. **Observe MinIO Console:**
    - Go to http://localhost:9001 and log in.
    - Click on the raw-data-bucket.
    - You will see the financial_data_delta/ and insurance_data_delta/ directories.

    - Periodically refresh or navigate into these directories. You will observe new .parquet files appearing, along with the _delta_log directory. Each .parquet file represents a micro-batch of data written by Spark.
2. Query Data (via Spark-SQL from within Spark container):
   You can directly query the data stored in MinIO using Spark's SQL interface.
   ```
   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 COUNT(*) FROM delta.\`s3a://raw-data-bucket/financial_data_delta\`;"
   ```

   Run this query multiple times over a few minutes.

Verification:

- **MinIO Console:** The presence of _delta_log and continuously appearing .parquet files within the financial_data_delta and insurance_data_delta paths confirms that Spark is successfully writing data to MinIO.
- **Spark-SQL Count:** The COUNT(*) query should show an increasing number of records, demonstrating continuous data ingestion and storage in MinIO.

Advanced Use Case 1: Hosting Delta Lake for ACID Transactions and Time Travel

Objective: To demonstrate how MinIO, coupled with Delta Lake, provides ACID transaction capabilities and time travel, making your S3-compatible data lake reliable and auditable.
Role in Platform: Elevate raw object storage into a transactional data lakehouse, enabling reliable data upserts, deletes, and historical querying.
Setup/Configuration:

1. **Ensure Basic Use Case is running:** financial_data_delta is populated with streaming data.
2. **pyspark_jobs/delta_merge_cdc.py:** This script (from "Highlighting Delta Lake" Advanced Use Case 3) demonstrates MERGE INTO operations and time travel, which rely on MinIO hosting the Delta Lake files.

Steps to Exercise:

1. **Stop financial_transactions streaming job:** If it's still running from previous examples, stop it (Ctrl+C in its terminal or docker compose stop spark).
2. **Run a Batch Update/Merge Job:**
    - Submit the pyspark_jobs/delta_merge_cdc.py job which performs an upsert (merge) operation on a Delta table hosted in MinIO.

```
docker exec -it spark spark-submit \
  --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 \
  /opt/bitnami/spark/jobs/delta_merge_cdc.py \
  s3a://curated-data-bucket/financial_transactions_dim
```

3. **Inspect MinIO after Merge:**
   - Navigate to http://localhost:9001, then curated-data-bucket/financial_transactions_dim/.
   - You'll see new .parquet files and an updated _delta_log reflecting the merge operation. The number of .parquet files might not directly correspond to the number of records due to Delta Lake's file compaction and versioning.
4. **Perform Time Travel Queries:**
   - Get the history of the table to find previous versions:

     ```
     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 "DESCRIBE HISTORY
     delta.\`s3a://curated-data-bucket/financial_transactions_dim\`;"
     ```

   - Note a VERSION number from *before* your last MERGE INTO operation.
   - Query the table "as of" that historical version:

     ```
     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 * FROM
delta.\`s3a://curated-data-bucket/financial_transactions_dim\` VERSION AS OF
<PREVIOUS_VERSION_NUMBER>;"
```

Replace <PREVIOUS_VERSION_NUMBER> with the actual version number.

Verification:

- **MinIO Console:** The _delta_log directory for financial_transactions_dim contains transaction files (e.g., 00000000000000000000X.json), demonstrating the transaction log.
- **Time Travel Query Results:** The query VERSION AS OF successfully retrieves the state of the data from a past version, confirming MinIO's capability to host Delta Lake tables with time travel enabled.

Advanced Use Case 2: Serving Curated Data for Analytical Consumption

Objective: To demonstrate how MinIO hosts highly optimized, curated Delta Lake tables, making them readily available for analytical tools like Spark SQL or other query engines (conceptually).

Role in Platform: Provide a performant and structured layer for data consumers (BI tools, data scientists, machine learning models) to access high-quality, transformed data.

Setup/Configuration:

1. **Ensure financial_data_curated_batch is populated:** Your batch transformation job (from "Highlighting Apache Spark" Advanced Use Case 1, pyspark_jobs/batch_transformations.py) should have written data to s3a://curated-data-bucket/financial_data_curated_batch.
2. **Spark ml_model_inference.py:** This script (from "Highlighting Apache Spark" Advanced Use Case 2) reads from the curated path, demonstrating a consumer.

Steps to Exercise:

1. **Inspect Curated Data in MinIO:**
   - Navigate to http://localhost:9001, then curated-data-bucket/financial_data_curated_batch/.
   - You will see the .parquet files (likely larger and fewer than in the raw zone due to compaction) and a _delta_log.
2. **Query Curated Data with Spark-SQL:**
   - Use Spark-SQL to directly query the curated Delta table.

```
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, amount, enriched_category, is_amount_valid, processing_timestamp FROM delta.\`s3a://curated-data-bucket/financial_data_curated_batch\` LIMIT 10;"
Observe: The data should be transformed, include enriched fields (like enriched_category), and is_amount_valid flags, reflecting the quality and structure expected for analytical consumption.

3. **Run a Conceptual ML Inference Job (reads from curated):**
    - Submit the pyspark_jobs/ml_model_inference.py job which reads from the curated path.

```
docker exec -it spark spark-submit \
  --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 \
  /opt/bitnami/spark/jobs/ml_model_inference.py \
  s3a://curated-data-bucket/financial_data_curated_batch
```

Observe: The job logs will confirm it's reading from the specified MinIO path and performing feature engineering.

Verification:

- **MinIO Console:** The curated-data-bucket contains the expected transformed data.
- **Spark-SQL Query:** The query results confirm the data is cleaned, enriched, and validated, ready for direct use by downstream analytical applications.
- **ML Inference Job:** Successful execution of the ML script demonstrates MinIO's role in providing structured, high-quality features for machine learning.

Advanced Use Case 3: Simulating S3 Event Notifications for Downstream Processes (Conceptual)

Objective: To conceptually demonstrate how MinIO can trigger event notifications (similar to AWS S3 Event Notifications) when new objects are created, which can then be consumed by other services to initiate downstream workflows.

Role in Platform: Enable event-driven micro-ETL or trigger specific processes (e.g., metadata ingestion, lightweight data validation Lambdas) as soon as new data lands in the data lake.

Setup/Configuration:

1. **Enable MinIO Event Notifications:** MinIO supports webhook notifications. You'd typically configure this in its startup command or configuration.
2. **Create a Dummy Webhook Listener:** For this demo, you'd need a simple HTTP server (e.g., a small Python Flask/FastAPI app) running in another container that acts as a webhook receiver.

Example docker-compose.yml snippet (conceptual additions for webhook listener):

```
# ...
webhook_listener:
  build: ./webhook_listener_app # A simple Flask/FastAPI app
  ports:
    - "8081:8081" # Expose to host
  # No healthcheck for simplicity
# ...
# MinIO service definition (add 'command' for events)
minio:
  image: minio/minio:latest
  # ... existing config
  command: server /data --console-address ":9001"
  # Add configuration for notifications (conceptual, often via mc client or startup script)
  # This part is typically done *after* MinIO starts, via `mc event add` command.
  # For a Docker Compose setup, you might have an entrypoint script that runs this.
  # E.g., `command: sh -c "minio server /data --console-address :9001 & mc alias set
local http://localhost:9000 minioadmin minioadmin && mc mb local/my-bucket-events
--ignore-existing && mc event add local/my-bucket-events arn:minio:sqs::1:webhook
--suffix .parquet --event put,delete"`
  # The `arn:minio:sqs::1:webhook` part is the important webhook target.
  # You'd need to configure the webhook target: mc admin config set
notify_webhook:webhook_target endpoint='http://webhook_listener:8081/minio-event'
queue_limit=100
```

Example webhook_listener_app/app.py (simple Flask app to receive webhooks):

```
# webhook_listener_app/app.py
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/minio-event', methods=['POST'])
def minio_event():
    event = request.json
    print("Received MinIO Event:")
    print(json.dumps(event, indent=2))
    # Process the event, e.g., trigger a Lambda, update a database
    return jsonify({"status": "success", "message": "Event received"}), 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8081, debug=True)
```

Steps to Exercise (Conceptual):

1. **Start Services:** Bring up minio, webhook_listener, and other necessary services.
2. **Configure MinIO Event Notification:**
    - This is the trickiest part to automate directly in docker-compose.yml. You'd typically use the mc (MinIO Client) after MinIO starts.
    - From your host machine, after minio container is up, execute:
      docker exec -it minio bash
      # Inside minio container:
      mc alias set local http://localhost:9000 minioadmin minioadmin
      mc mb local/my-event-bucket --ignore-existing # Create a bucket for events
      mc event add local/my-event-bucket arn:minio:sqs::1:webhook --suffix .parquet --event put # Add put object event
      mc admin config set notify_webhook:webhook_target endpoint='http://webhook_listener:8081/minio-event' queue_limit=100 --console-address ":9001"
      # Exit the minio container.

    - Restart MinIO service in docker-compose if changes to eventing require it. docker compose restart minio
3. **Upload a File to the Event Bucket:**
    - Go to http://localhost:9001, navigate to my-event-bucket, and manually upload a .parquet file (or any file with the configured suffix).
    - Or, use mc cp <local_file.parquet> local/my-event-bucket/.
    - Alternatively, configure a Spark job to write to s3a://my-event-bucket/.
4. **Observe Webhook Listener Logs:** Watch the logs of your webhook_listener container.

Verification (Conceptual):

- **Webhook Listener Logs:** The webhook_listener container's logs will show the JSON payload of the MinIO event, containing details about the object put operation (bucket name, object key, timestamp, etc.). This demonstrates MinIO's ability to emit events for new objects, enabling event-driven architectures.
- This feature is crucial for implementing patterns like serverless ETL where a Lambda function triggers upon new file arrival in S3.

This concludes the guide for MinIO.