Tick Data Machine Learning Trading Pipeline: System Architecture & Technical Documentation

Foreword

This document provides a high-level introduction and structural overview of the **Tick Data Machine Learning Trading Pipeline (JAN TO FEB 2026)**.

The system is designed to ingest high-frequency financial tick data, process and store it efficiently, and apply machine learning techniques—specifically anomaly detection—to support quantitative analysis, backtesting, and live trading execution.

The pipeline follows a modular and scalable architecture, separating concerns across data ingestion, data processing, storage, machine learning, backtesting, and deployment. Emphasis is placed on reproducibility, data integrity, version control, and experiment tracking to ensure reliable research and production-grade execution.

This document serves as the **entry point** to the full technical documentation set. Detailed implementation notes, configuration references, and experimental results are maintained separately and compiled into the final documentation bundle.

Here is the pipeline diagram for further context:

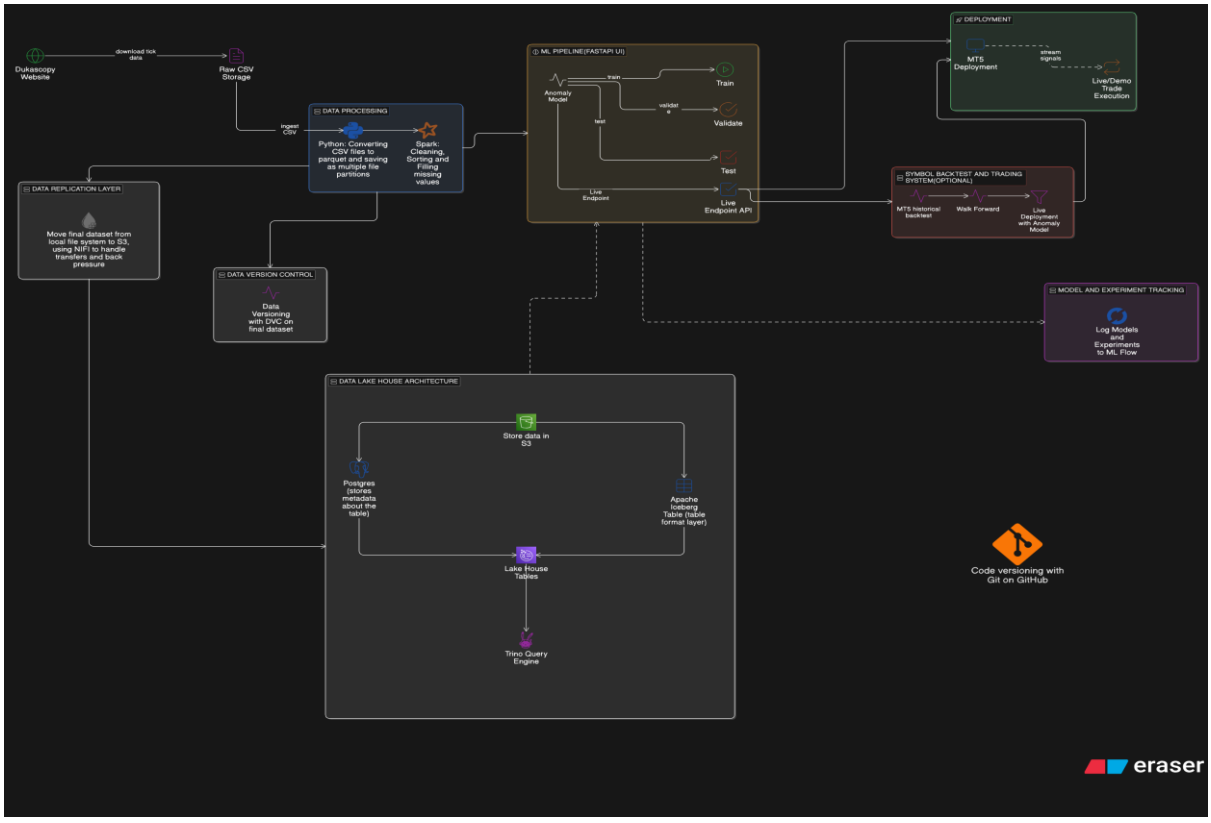

Table of Contents

Tick Data Acquisition

The first stage of the pipeline is responsible for the acquisition of **raw, high-frequency tick data** from the **Dukascopy** market data service using **Quant Data Manager**.

Quant Data Manager is used as a lightweight and straightforward data collection application, providing a simple interface for downloading historical tick data from the Dukascopy website. A **free version** of the software is available, making it suitable for individual research and prototyping use cases. The application is **officially available for Windows platforms**; support for macOS was not a requirement for this pipeline and was not evaluated.

Tick data is collected at the finest available temporal resolution and includes individual market events such as bid and ask price updates, Date and Time, and associated volumes. During the download process, the data is written directly to disk as **CSV files**, with no transformation, aggregation, or filtering applied.

The resulting CSV files represent the **raw, immutable data layer** of the pipeline. Once saved, these files are treated as read-only and serve as the authoritative source for all downstream processing, validation, and analysis. Preserving the data in its original CSV form ensures traceability, reproducibility, and the ability to reprocess the dataset as pipeline logic evolves.

Quant Data Manager is used exclusively for data acquisition, while all subsequent processing, storage optimization, and machine learning workflows are handled in later stages of the system. The application itself is easy to use and setup, if one wishes to collect data the manually way using a web scrapper, best of luck as it will be a very time-consuming task, **QDM** simply makes all this easier.

For the next step, this data will be sent over to processing scripts that will clean, sort and handle all kind of data fragmentation issues, if there were any to begin with.

Data Processing

Overview

The project consists of a suite of scripts designed to process, clean, and analyze high-frequency tick data. The scripts operate sequentially, with the output of one stage serving as the input to the next.

The scripts included in this pipeline are:

- **csv_to_parquet_converter.py**
  Converts CSV files to Parquet format in a single-threaded manner.
- **spark_csv_parquet_converter_ver_1.py**
  Converts CSV files to Parquet format using Apache Spark for improved performance.
- **spark_data_cleaner_script.py**
  Scans Parquet files for data quality issues, reporting null values and corrupted files.
- **spark_session_window_tick_cleaner.py**
  Aggregates tick data into 1-second windows and saves the cleaned output.
- **symbol_statistics_generate.py**
  Generates daily tick count statistics for specified time windows and produces plots.

Each script has clearly defined paths, configurations, and processing logic for handling, cleaning, and analyzing tick data.

1. csv_to_parquet_converter.py

Purpose

This script converts CSV files containing tick data into Parquet format using the **PyArrow** library. Data is processed in chunks to efficiently handle large file sizes.

Key Components

Constants

- `INPUT_BASE`: Base directory containing input CSV files
- `OUTPUT_BASE`: Directory for saving output Parquet files
- `CSV_CHUNK_SIZE`: Size of CSV chunks (500 MB)
- `CSV_SCHEMA`: Schema definition for incoming data

Functions

- `already_processed(csv_path)`
  Checks whether a CSV file has already been converted to avoid duplicate processing.
- `process_csv(csv_path)`
  Reads a CSV file, removes the `Volume` column, and writes the data to Parquet format in chunks.
- `process_all()`
  Iterates through all directories and CSV files under `INPUT_BASE` and processes each file.

Usage

The script is executed directly. Files are processed strictly one at a time, and progress updates are printed to the console.

2. spark_csv_parquet_converter_ver_1.py (Alternative version)

Purpose

This script improves CSV-to-Parquet conversion performance by leveraging **Apache Spark** for batch processing.

Key Components

Constants

- `INPUT_BASE`: Directory containing input CSV files
- `OUTPUT_BASE`: Directory for output Parquet files
- `INPUT_SPLIT_SIZE`: Maximum input split size (500 MB)
- `PARQUET_BLOCK_SIZE`: Parquet block size (512 MB)

Spark Initialization

- Initializes a Spark session with performance-oriented configurations.

Functions

- `process_directory(input_dir: Path)`
  Reads CSV files from a directory, drops the `Volume` column, and writes the data to Parquet format.

Usage

The script recursively processes directories under `INPUT_BASE`, converting all CSV files to Parquet format.

3. spark_data_cleaner_script.py

Purpose

This script performs data quality checks on Parquet files generated by earlier stages. It identifies corrupted files and reports null values in key columns.

Key Components

Constants

- `INPUT_BASE`: Directory containing Parquet files
- `REPORT_PATH`: Path for saving the summary report

Functions

- `scan_parquet_file(parquet_file)`
  Loads a Parquet file, counts total rows, identifies null values, and detects loading errors.

Process

- Scans all Parquet files
- Collects quality metrics per file
- Saves a consolidated report in CSV format

Usage

Run the script directly to generate a data integrity report for all processed Parquet files.

4. spark_session_window_tick_cleaner.py

Purpose

This script aggregates tick data stored in Parquet format into **1-second time windows** and saves the cleaned output.

Key Components

Constants

- `BASE_INPUT_FOLDER`: Directory containing input Parquet files
- `BASE_OUTPUT_FOLDER`: Directory for cleaned output files
- `MAX_PARTITION_BYTES`: Spark partition size configuration

Functions

- Helper functions for timestamp parsing
- File ordering based on timestamps
- Aggregation logic for tick data

Process

- Reads all relevant Parquet files
- Filters data by specified time ranges
- Aggregates ticks into 1-second windows
- Writes cleaned data as single Parquet files

Usage

Execute the script directly to process all symbols found in the input directory and generate cleaned tick data.

5. symbol_statistics_generate.py

Purpose

This script generates daily tick count statistics for specified time windows and produces visual plots.

Key Components

Constants

- `base_path`: Location of processed Parquet files
- `output_path`: Destination for reports and plots

Functions

- `filter_window(df, start, end)`
  Filters records within a specified time window.

Process

- Iterates through symbol directories
- Filters data by time window
- Aggregates tick counts by day
- Saves results as CSV files and plots

Usage

Run the script to generate both numerical reports and visual representations of daily tick activity per symbol.

Conclusion

This suite of scripts provides an end-to-end workflow for converting raw tick data from CSV to Parquet format, validating data quality, aggregating time-based data, and generating analytical reports and visualizations.

The scripts are intended to be executed sequentially, with each stage building upon the outputs of the previous stage, ensuring a structured, reproducible, and efficient data processing pipeline. In the next step, we will explore the machine learning architecture and see how we can use this processed data to make live inference.

ML Pipeline

The ML pipeline is divided into several modules that handle various stages of the machine learning workflow, from data loading to model training and validation, to serving the model via FastAPI and monitoring its health. Below is a detailed description of each module with their corresponding classes and methods.

Note: Please ensure you run main.py to start the fastapi UI page, also in another terminal run the mlflow UI so you can view the experiments and access the model and its parameters. The automate_pipeline.py is merely optional, you can skip straight to the main.py script if automation is not your kind of thing, but remember you wll need to manually call the endpoints on the UI.

1. automate_pipeline.py

Overview

This module provides a script to automatically run the entire ML pipeline. It includes functionalities such as checking if the FastAPI server is running, starting training, validation, and testing, and managing the live endpoint.

Classes

PipelineAutomator

The PipelineAutomator class encapsulates the whole automation pipeline.

Methods:

- `__init__()`: Initializes the automator with default parameters and paths.
- `check_fastapi_running()`: Checks if the FastAPI server is running by making a GET request.
- `wait_for_fastapi(max_wait=120)`: Waits for a specified duration for the FastAPI server to start by repeatedly checking its status.
- `start_train_val()`: Initiates the training and validation phase by sending a POST request to the FastAPI endpoint.
- `start_test()`: Initiates the testing phase through a POST request.
- `start_live_endpoint()`: Sends a request to start the live endpoint for real-time predictions.

- `check_live_status()`: Checks the status of the live endpoint.
- `wait_for_live_activation(max_wait=60)`: Waits for the live endpoint to become fully activated.
- `get_status()`: Retrieves the overall pipeline status via a GET request.
- `wait_for_completion(phase="train_val", check_interval=10)`: Monitors whether a specified phase (training, validation, or testing) has completed.
- `load_parameters()`: Loads model parameters from a JSON file.
- `ask_auto_live_setting()`: Prompts the user whether to automatically start the live endpoint after testing.
- `run_pipeline()`: Executes the complete ML pipeline, sequentially performing training, testing, and live deployment.

Main Function

- `main()`: The entry point of the script where the PipelineAutomator is instantiated and the pipeline is executed.

2. data_loader.py

Overview

The data_loader.py module is responsible for loading tick data from specified file paths. It supports loading data for individual symbols or all symbols defined in a symbols list.

Classes

SimpleTickLoader

A class designed to load tick data for a set of trading symbols.

Methods:

- `__init__(symbols_path: str, data_folder: str)`: Initializes the loader with paths to the symbols JSON file and the data folder.
- `load_symbols()`: Loads the list of symbols from the specified JSON file.
- `load_one(symbol: str) -> pd.DataFrame`: Loads data for a single symbol.
- `load_some(symbols: list) -> pd.DataFrame`: Loads data for multiple specified symbols.

- `load_all() -> pd.DataFrame`: Loads data for all symbols defined in the symbols path.
- `check_symbols()`: Checks which symbols have corresponding data and returns a summary.

Quick Load Functions

- `load_symbol(symbol: str) -> pd.DataFrame`: Loads data for a single symbol.
- `load_symbols(symbols: list) -> pd.DataFrame`: Loads data for multiple symbols.
- `load_all_symbols() -> pd.DataFrame`: Loads data for all available symbols.

3. fastapi_utils.py

Overview

This module contains utility functions used by the FastAPI application to load parameters, validate configuration, and manage paths.

Functions

- `load_parameters(json_path: str) -> Dict`: Loads parameters from a JSON file.
- `get_paths_from_params(params: Dict) -> Dict[str, Path]`: Extracts paths from the loaded parameters and validates their existence.
- `validate_parameters(params: Dict) -> bool`: Validates the presence of required sections in the parameters.
- `get_trained_symbols(artifacts_path: Path) -> list`: Retrieves a list of symbols whose models have been trained.
- `create_response(success: bool, message: str, data: Dict = None) -> Dict`: Constructs a standardized response format for API requests.

4. main.py

Overview

The main FastAPI application module that combines all pieces, handling the API requests for training, testing, and serving the live endpoint.

Global Variables

- `pipeline_state`: Holds the current state of training, testing, and live endpoint status.

API Endpoints

- `@app.get("/")`: Root endpoint returning information about the service.
- `@app.post("/train_val")`: Starts the training and validation process.
- `@app.post("/test")`: Initiates the model testing phase.
- `@app.post("/live/start")`: Starts the live endpoint for predictions.
- `@app.post("/live/stop")`: Stops the running live endpoint.
- `@app.get("/live/health")`: Checks the health status of the live endpoint.
- `@app.get("/status")`: Retrieves the current status of the entire pipeline.
- `@app.get("/docs")`: Customizes and serves the Swagger documentation for the API.

5. mlflow_utils.py

Overview

This module tracks experiments using MLflow, logging various metrics, parameters, and models during training and validation phases.

Classes

MLflowPhaseTracker

A class to track the machine learning phases within MLflow.

Methods:

- `__init__(symbol: str, experiment_name_prefix: str = "TickAnomaly", tracking_uri: Optional[str] = None)`: Initializes the tracker for a specific symbol.
- `setup_experiment()`: Creates or accesses an MLflow experiment for the given symbol.
- `start_phase_run(phase: str, parent_run_id: Optional[str] = None, tags: Optional[Dict] = None) -> str`: Starts a new MLflow run.
- `end_phase_run()`: Ends the current active run.
- `log_phase_params(params: Dict)`: Logs parameters for the current phase.
- `log_phase_metrics(metrics: Dict, step: Optional[int] = None)`: Logs metrics for the current phase.
- `log_phase_artifact(local_path: str, artifact_path: Optional[str] = None)`: Logs an artifact for the current phase.
- `log_pytorch_model(model: nn.Module, model_name: str, input_example_np: np.ndarray, metadata: Optional[Dict] = None)`: Logs a PyTorch model to MLflow.

6. model.py

Overview

Defines the core model architecture (Autoencoder) and includes functionalities to prepare and handle window-based data.

Classes

WindowTickDataDataset

A PyTorch Dataset for window-based tick data.

Methods:

- `__init__(data_dict, features, ticks_per_window=600)`: Initializes the dataset with window data.

Autoencoder

A basic autoencoder model for anomaly detection.

Methods:

- `__init__(input_dim, hidden_dims=[256, 128, 64], latent_dim=16, ...)`: Initialization and setup of the autoencoder architecture.
- `forward(x)`: Defines the forward pass of the model.
- `get_reconstruction_error(x)`: Computes the reconstruction error for the given input.

Functions

- `filter_time_windows(df)`: Filters the DataFrame to keep data only during specified time windows.
- `compute_spread(df)`: Computes the spread from Bid and Ask prices and adds it as a new column.
- `prepare_tick_features(df)`: Prepares the dataset by cleaning and vectorizing required features.
- `create_window_based_data(df, features, ticks_per_window=600)`: Creates window-based datasets from tick data.
- `prepare_window_data(df, features, ticks_per_window=600)`: Prepares window data, padding it appropriately.
- `split_window_data(window_data, train_ratio=0.6, val_ratio=0.2, test_ratio=0.2)`: Splits window data into train, validation, and test sets while maintaining the specified ratios.
- `scale_window_datasets(train_data, val_data, test_data, features)`: Scales datasets using StandardScaler.

7. train.py

Overview

The main training and validation script that processes each symbol independently, manages the training loop, and integrates with MLflow for tracking.

Functions

- `main_train_val(params: Dict, paths: Dict[str, Path])`: Main function to execute training and validation for all specified symbols.
- `process_symbol_train_val(symbol: str, df_symbol: pd.DataFrame, params: Dict, artifacts_path: Path) -> bool`: Processes individual symbols for training and evaluates the model.

Classes

SymbolTrainer

Handles the training and validation process for each symbol.

Methods:

- `prepare_data()`: Prepares the tick data and creates datasets for training, validation, and testing.
- `create_model()`: Constructs the Autoencoder model according to specified parameters.
- `run_training_phase()`: Executes the training loop and implements early stopping if enabled.
- `run_validation_phase()`: Validates the model after training to evaluate its performance.
- `save_artifacts()`: Saves necessary artifacts such as model weights, training curves, and configuration details.

8. test.py

Overview

`test.py` is a testing script for evaluating the performance of a trained window-based autoencoder model. It loads pre-trained models, tests them against available test datasets, computes reconstruction errors, and optionally logs results in MLflow.

Functions

- `main_test(params: Dict, paths: Dict[str, Path]) -> Dict`: Orchestrates the testing process for all trained autoencoder models. Raises `ValueError` if no trained models are found.
- `process_symbol_test(symbol: str, artifacts_path: Path, params: Dict) -> bool`: Tests a single trained autoencoder model, handling model loading, test data execution, and exceptions.

Classes

SymbolTester

Manages the testing processes for a single symbol's autoencoder model.

Methods:

- `__init__(self, symbol: str, artifacts_path: Path, params: Dict, mlflow_tracker=None)`: Initializes the class with necessary configurations.
- `get_or_create_config(self)`: Loads the model configuration or creates it from parameters.
- `load_model(self)`: Loads the trained autoencoder model and its weights.
- `load_test_data(self)`: Loads test data and scaler objects.
- `run_testing_phase(self)`: Executes testing and logs results.

Summary
`main_test` saves a summary JSON file with counts of successful and failed tests.

9. broker_symbols.py

Overview

Contains a class that maps generic symbols used in the model to broker-specific symbols. This enables compatibility with various trading platforms.

Classes

BrokerSymbols

Methods:

- `get_broker_name() -> str`: Retrieves the broker server name and determines the relevant broker.
- `get_broker_symbol(generic_symbol: str) -> str`: Returns the broker-specific symbol.
- `get_all_broker_symbols(generic_symbols: list) -> dict`: Converts a list of generic symbols into broker-specific symbols.
- `print_broker_info()`: Prints current broker information for debugging.

Symbol Mappings
A class-level dictionary `SYMBOL_MAPPINGS` contains mappings for different brokers.

10. live_endpoint.py

Overview

Implements a live trading monitor that captures tick data, processes it through the autoencoder model, and generates predictions in real-time.

Main Class: LiveEndpointManager

Initialization
Loads symbols from a JSON file and sets up logging.

Core Methods:

- `start() -> bool`: Starts the live endpoint, loads models, and starts processing.
- `stop() -> bool`: Safely stops the live endpoint and joins the thread.
- `health_check()`: Provides a summary status including predictions and detected anomalies.

Private Methods:

- `_load_models()`: Loads models for the specified symbols and handles errors.
- `_init_mt5()`: Initializes MT5 connection and selects required symbols.
- `_run_live_loop()`: Main loop for collecting ticks and generating predictions.
- `_check_window_transitions()`: Checks transitions between trading windows and triggers actions.

Tick Processing

The `TickProcessor` class processes incoming tick data, prepares it for predictions, and computes results in real-time.

Conclusion
This module provides a framework for live anomaly detection and periodic testing of model performance. Its modular design allows easy extension and modification. For advanced features like Algorithmic trading, you will need to build your own Expert

Advisor in MT5, and do all kind of backtest, forward testing as well as some demo trading before going live. At best this model will only give signals and based on those signals, you can decide on what kind of strategy you can implement. THIS IS BY NO MEANS FINANCIAL ADVICE!!!

Next, we shall look external architectures like Data Lake houses, Apache Nifi and data replication to cloud as well as data versioning using DVC. It is basically like Git but for datasets, there will not be any documentation for it, however, feel free to check my YouTube playlist on how DVC can be used (link on my GitHub repo for this project). Our focus will be on the Data lake house and replication layer, for backup and storage optimizations.

NOTE: Scripts such as live_endpoint.py and broker_symbols.py will not be provided as they can vary between brokers and trading platforms, they are left out of the repo to ensure everyone is able to make their own scripts to cater for their own needs.

Data Replication Layer with Apache Nifi

Our initial setup for the data replication layer will consist of two docker-compose.yml files, both of which are important for this step and the next step in the data lakehouse.

Preferably, both YAML files can be combined into a single file to avoid networking or dependency issues, but the system also works correctly when they are deployed separately.

Since both YAML files are already explained in this section, the full data lakehouse setup will not be repeated again later. From this point forward, the focus will be on the creation of Iceberg tables, time travel functionality, and partitioning strategies.

Overview

This environment consists of two related Docker Compose configurations stored in separate folders. The first folder contains the NiFi deployment. The second folder, named Apache Iceberg, contains the Trino, MinIO, and PostgreSQL stack.

Together, these components form a complete data lakehouse architecture where NiFi handles ingestion, MinIO provides object storage, Trino performs SQL querying, and PostgreSQL stores Iceberg metadata. The two compose files are connected through shared Docker networks so the services can communicate properly.

Network Design

Two Docker networks are used in this setup. Each has a specific purpose and controls how services communicate.

The first network is called datalake-net. This is an external network that must be created manually before starting the containers. Its main purpose is to allow NiFi to communicate with MinIO. This acts as the ingestion-to-storage bridge where NiFi sends data into the object store.

The second network is called trino-net. This is an internal bridge network created automatically by Docker Compose. It allows Trino to communicate with both MinIO and PostgreSQL. Trino reads data from MinIO and reads or writes Iceberg metadata in PostgreSQL.

MinIO is connected to both networks. This makes it the central storage layer. It receives data from NiFi through datalake-net and serves data to Trino through trino-net.

NiFi Stack (First Docker Compose File)

This configuration deploys an Apache NiFi instance configured for secure web access, persistent storage, and access to a local training data folder.

The service uses the apache/nifi image and exposes port 8443 so the interface can be accessed at https://localhost:8443.

Environment variables define the HTTPS port, the single-user login credentials, and the proxy host used to access the NiFi web interface. The password must be at least twelve characters long.

Several host directories are mounted into the container to ensure persistence. These directories store the database repository, flowfile repository, content repository, provenance repository, state data, and logs. This ensures that flows and history are not lost when the container restarts.

A custom bind mount is also used to map a local Training Batch folder into the container at /data/training_batch. This allows NiFi to read local files and ingest them into MinIO.

The NiFi container connects to the datalake-net network so it can communicate directly with the MinIO service. The restart policy is set to unless-stopped so the service automatically restarts after crashes or system reboots.

Iceberg Stack (Second Docker Compose File in the Apache Iceberg Folder)

This configuration deploys three services: Trino, MinIO, and PostgreSQL. Together they provide the analytics and metadata layer for the lakehouse.

Trino Service

The Trino service acts as the distributed SQL query engine. It uses the trinodb/trino image and exposes port 8081 on the host mapped to port 8080 inside the container. This allows access to the Trino interface at http://localhost:8081.

A volume is mounted from the host into /etc/trino/catalog. This directory contains catalog configuration files used to connect Trino to Iceberg, MinIO, and PostgreSQL.

The service is configured to wait until MinIO and PostgreSQL are healthy before starting. This ensures the query engine does not start before its dependencies are ready.

Java memory settings are provided using JVM options to allocate one gigabyte as the starting memory and up to two gigabytes maximum.

Trino connects to the trino-net network so it can communicate with both MinIO and PostgreSQL.

MinIO Service

MinIO provides S3-compatible object storage. It stores raw ingested data, parquet files, and Iceberg table data.

The service uses the minio/minio image and exposes two ports. Port 9000 is used for the S3 API and port 9001 is used for the web console. The console can be accessed at http://localhost:9001.

Environment variables define the root username and password for logging into the MinIO console.

The startup command initializes the server using the /data directory for storage and binds the console interface to port 9001.

A health check runs every five seconds to confirm that the service is ready.

MinIO connects to both datalake-net and trino-net. This allows it to receive data from NiFi and provide data to Trino at the same time.

PostgreSQL Service

PostgreSQL acts as the Iceberg catalog database. It stores metadata such as table definitions, namespaces, schema versions, and snapshots.

The service uses the postgres:14 image and is assigned the container name iceberg-postgres.

Environment variables define the database name, user, and password.

Port 5433 on the host is mapped to port 5432 in the container, allowing local access to the database.

A named volume is used to persist database files so data is not lost when the container restarts.

An initialization SQL script is mounted and automatically executed during first startup to create the required Iceberg catalog tables.

A health check ensures PostgreSQL is ready before Trino attempts to connect.

PostgreSQL connects only to the trino-net network because only Trino needs access to the metadata database.

Data Flow Explanation

The system follows a clear flow from ingestion to analytics.

First, NiFi reads local training batch data and uploads it into MinIO buckets.

Second, MinIO stores the data as objects in S3-compatible storage.

Third, Trino creates Iceberg tables that point to data stored inside MinIO.

Fourth, PostgreSQL stores all Iceberg metadata such as schemas and table structure.

Finally, Trino queries the data by combining object storage from MinIO with metadata stored in PostgreSQL.

Why Two Separate Docker Compose Files?

Keeping the ingestion stack and analytics stack separate provides better modularity and flexibility.

NiFi can run independently without affecting the analytics components. The Iceberg stack can also be restarted or scaled separately. This separation improves troubleshooting, maintenance, and overall architecture clarity.

Deployment Order

First, create the shared external network by running the command to create datalake-net.

Second, navigate to the Apache Iceberg folder and start the stack so that MinIO, PostgreSQL, and Trino come online.

Third, start the NiFi stack. Once running, NiFi will be able to connect to MinIO through the shared datalake-net network.

Creating Nifi Process Group(s)

Once the Nifi container is up and running, you will need to access the UI, provide your credentials and log it. Then head straight to Process Group on the the top panel alongside with the other icons. Click on the icon and drag it onto the canvas, a new window will appear, click on the icon on the right side of the textbox, and you will be able to browse for your files.

Look for the JSON file that was in my GitHub repo and select it, this file will create our Nifi processors on the canvas automatically.

Click add then click on the Nifi Flow, then a new canvas will open with all the processors.

Next you want to enable all controller services, right click and enable controller service. Please note that you may have to retype your S3 bucket credentials, so Nifi is able to copy the files from your local machine into S3. Then manually start the processors manually, for better handling, start them backwards to avoid tasks from piling up in the connection queues, start off with PutS3Object going backwards.

Should this step be too complex, the YouTube video on my channel shall handle and clear up any confusion.

Summary

This setup creates a complete local data lakehouse environment. NiFi handles ingestion, MinIO acts as the central object store, Trino provides distributed SQL querying, and PostgreSQL stores Iceberg metadata.

All services are connected through well-defined Docker networks that allow secure communication, persistent storage, and reliable startup behavior.

Next we we head over to our S3 bucket in minio, to setup the datalake house and all its features.

Data Lake House Architecture (Gold)

With the replication layer in place, the environment is now ready for the data lakehouse operations. This layer integrates object storage, a metadata catalog, and a distributed query engine to provide a unified platform for analytical workloads.

At this stage, the data is “golden” and prepared for machine learning pipelines. The focus will be on:

- Organizing data in object storage for efficient access
- Creating and managing Iceberg tables in the metadata catalog
- Querying and exploring the datasets using the distributed query engine
- Applying partitioning strategies and leveraging time travel for reproducibility

This ensures that the data is structured, consistent, and ready for downstream ML and analytics tasks.

Overview

This section showcases the scripts needed to setup the Apache Iceberg tables, metadata level and rollback abilities, the script are displayed in the correct order so you can run them without issues.

1. init-iceberg.sql

This SQL script initializes the Iceberg JDBC catalog tables within Trino. It creates various tables necessary for Iceberg's metadata management and sets up the necessary schema and properties to effectively store and retrieve data.

Tables Created

- **`iceberg_namespace_properties`**
  - **Purpose**: Stores properties for namespaces (catalogs).
  - **Columns**:
    - `catalog_name`: Name of the catalog (VARCHAR(255), NOT NULL).
    - `namespace`: The namespace (or schema) within the catalog (VARCHAR(255), NOT NULL).
    - `property_key`: Key for the property (VARCHAR(255), NOT NULL).
    - `property_value`: Value of the property (TEXT).
  - **Primary Key**: (`catalog_name`, `namespace`, `property_key`).

- **`iceberg_tables`**
  - **Purpose**: Contains metadata about Iceberg tables.
  - **Columns**:
    - `catalog_name`: Catalog name (VARCHAR(255), NOT NULL).
    - `table_namespace`: Namespace for the table (VARCHAR(255), NOT NULL).
    - `table_name`: Name of the table (VARCHAR(255), NOT NULL).
    - `metadata_location`: Location of the metadata for the table (TEXT).
    - `previous_metadata_location`: Location of the previous metadata (TEXT).
  - **Primary Key**: (`catalog_name`, `table_namespace`, `table_name`).
- **`iceberg_columns`**
  - **Purpose**: Information on columns within tables.
  - **Columns**:
    - `catalog_name`, `table_namespace`, `table_name`: Identifiers for the table (similar to above).
    - `column_name`: Name of the column (VARCHAR(255), NOT NULL).
    - `column_type`: Data type of the column (VARCHAR(255), NOT NULL).
    - `column_position`: Position of the column in the table (INTEGER, NOT NULL).
  - **Primary Key**: (`catalog_name`, `table_namespace`, `table_name`, `column_name`).
- **`iceberg_partitions`**
  - **Purpose**: Holds partition information for tables.
  - **Columns**:
    - `catalog_name`, `table_namespace`, `table_name`: Identifiers for the table (similar to above).
    - `partition_spec_id`: The specification identifier for the partition (INTEGER, NOT NULL).
    - `partition_key`: Key value of the partition (VARCHAR(255), NOT NULL).
    - `transform`: Transformation applied to the partition (VARCHAR(255), NOT NULL).
    - `column_name`: Name of the column associated with the partition (VARCHAR(255), NOT NULL).
  - **Primary Key**: (`catalog_name`, `table_namespace`, `table_name`, `partition_spec_id`, `partition_key`).
- **`iceberg_snapshots`**
  - **Purpose**: Keeps track of table snapshots.

    - **Columns**:
        - `catalog_name`, `table_namespace`, `table_name`: Identifiers for the table (similar to above).
        - `snapshot_id`: Unique identifier for the snapshot (BIGINT, NOT NULL).
        - `parent_snapshot_id`: ID of the parent snapshot (BIGINT).
        - `timestamp_ms`: When the snapshot was taken (BIGINT, NOT NULL).
        - `manifest_list`: List of manifest files associated with the snapshot (TEXT).
        - `summary`: Summary of changes in the snapshot (TEXT).
    - **Primary Key**: (`catalog_name`, `table_namespace`, `table_name`, `snapshot_id`).
- **`iceberg_manifest_files`**
    - **Purpose**: Manages details about manifest files in snapshots.
    - **Columns**:
        - `catalog_name`, `table_namespace`, `table_name`: Identifiers for the table (similar to above).
        - `snapshot_id`: ID of the associated snapshot (BIGINT, NOT NULL).
        - `manifest_path`: Path to the manifest file (TEXT, NOT NULL).
    - **Primary Key**: (`catalog_name`, `table_namespace`, `table_name`, `snapshot_id`, `manifest_path`).
- **`iceberg_data_files`**
    - **Purpose**: Tracks data files linked with snapshots.
    - **Columns**:
        - `catalog_name`, `table_namespace`, `table_name`: Identifiers for the table (similar to above).
        - `snapshot_id`: ID of the snapshot (BIGINT, NOT NULL).
        - `file_path`: Path to the data file (TEXT, NOT NULL).
    - **Primary Key**: (`catalog_name`, `table_namespace`, `table_name`, `snapshot_id`, `file_path`).
- **`iceberg_refs`**
    - **Purpose**: Stores references to specific snapshots or tags.
    - **Columns**:
        - `catalog_name`, `table_namespace`, `table_name`: Identifiers for the table (similar to above).
        - `ref_name`: Name of the reference (VARCHAR(255), NOT NULL).
        - `snapshot_id`: Associated snapshot ID (BIGINT).
        - `type`: Type of the reference (VARCHAR(255), NOT NULL).
    - **Primary Key**: (`catalog_name`, `table_namespace`, `table_name`, `ref_name`).

- `iceberg_properties`
  - **Purpose**: Contains custom properties set for tables.
  - **Columns**:
    - `catalog_name`, `table_namespace`, `table_name`: Identifiers for the table (similar to above).
    - `property_key`: The property name (VARCHAR(255), NOT NULL).
    - `property_value`: Value of the property (TEXT).
  - **Primary Key**: (`catalog_name`, `table_namespace`, `table_name`, `property_key`).

Other Operations

1. **Create Schema**: Defines a schema named `gold`.
2. **Create Indexes**: To optimize performance on relevant tables.
3. **Insert Initial Namespace**: Sets an initial namespace properties entry for the catalog.

2. iceberg_migration.py

This Python script facilitates the migration and registration of raw parquet files from MinIO into the Iceberg format using Trino as the SQL interface. It automates tasks such as connecting to Trino, ensuring schemas exist, loading data, and managing tables.

Core Functionalities:

- **Configuration**: Define constants for connecting to Trino and MinIO.
- **MinIO Client**: Function `get_minio_client()` returns a MinIO client instance.
- **Trino Connection**: `get_trino_connection()` establishes a connection to Trino while handling retries on failure.
- **Auto Detection**: Functions to detect available symbols in the `Training Batch/` directory in MinIO.
- **Hive Table Registration**: Functions allow creation of Hive tables based on symbols detected and ensure that schemas and tables exist.
- **Iceberg Table Creation**: Create Iceberg tables pointing to the gold path in S3, and manage loading data from Hive to the Iceberg structure.
- **Data Loading**: Load data from registered Hive tables into corresponding Iceberg tables and enforce that data integrity is maintained.
- **Partition Verification**: The script can query partition counts per year after loading data.

3. metadata_layer.py

This script focuses on setting up the necessary metadata environment for managing Iceberg tables via Trino.

Core Functionalities:

- **Connection Management**: Functions for establishing connections to Trino and MinIO, with retry logic for stability.
- **Table Creation**: Verifies the existence of the gold schema and creates tables for each symbol found in MinIO.

4. iceberg_time_travel.py

This script provides functionalities to utilize Iceberg's time travel capabilities, enabling users to query data states from past snapshots and manage rollback features.

Core Functionalities:

- **Establish Connection**: Ensure a fresh connection to Trino that forces reloading of the current Iceberg snapshot pointer.
- **Snapshot Management**: Queries and retrieves snapshots and enables users to view their associated metadata, including additions and deletions between snapshots.
- **Partition Information**: Can retrieve partition data to assist in understanding the current or historical state of a table.
- **Rollback Functionality**: Allows rollback to a previous snapshot, refreshing the view to ensure the rollback is applied.
- **Interactive Menu**: Provides a user-friendly interface for selecting snapshot operations and executing queries based on user input.

Conclusion

The integrated scripts provide a comprehensive setup for managing an Iceberg data lakehouse. Key functionalities include schema management, data loading, and snapshot-based querying. Each component is designed to work seamlessly with Trino and MinIO, facilitating efficient data management and retrieval.

The provided documentation ensures that users and developers have a clear understanding of the code's purpose, functionality, and usage, enabling effective integration and maintenance.

This concludes the end of the documentation, for further feedback and assistance you are more than welcome to contact me.

https://www.linkedin.com/in/nokul-debanath-750a56352

AlgoDeveloper400

https://www.kaggle.com/nokuldebanath

https://www.youtube.com/@BDB5905