# Witside Technical Assessment pipeline

by R. N. Caballero

## 1) OVERVIEW

I created a pipeline in order to address the assessment described in the accompanying file “Witside_Assessment_and_Problem_Description.md”

The chosen route was the combined use of Python and SQL via SQLAlchemy.

Two (2) SQL scripts create the schema Tables and Views. The database creation, rest of calculations and report creation are executed in the Python code, which consists of consists of six (6) .py modules.

The pipeline was written in a way to simulate a professional business environment, running on local computing (so that it could be tested running on my PC). Therefore, the pipeline was built with automation, modularity, and scalability for future data and/or business questions scaling up or changing.

## 2) CONTEXT AND BUSINESS QUESTIONS

I am provided with a csv file from a production floor composed with multiple production lines. Here I go through the business questions asked and my interpretation of the questions in relation to the data.

Q1: For production line "gr-np-47", give me table with columns:
1) start_timestamp: the timestamp with the initiation of the production process.
2) stop_timestamp: the timestamp with the termination of the production process after the last initiation.
3) duration: the total duration of the production process.

-> Q1 can only be answered for production lines that have START and STOP in the provided data. The code can accept any production_line_id to check. For gr-np-47 this is possible and the result information was recovered.
##

Q2: What is the total uptime and downtime of the whole production floor?

-> For uptime/downtime, there are two ways to respond. A) Total up/down-time recorded in the period covered by the provided data. In this case all time spent in ON status (incl. time from a START to an ON status), is added to uptime. Respectively for downtime I consider time spent on STOP until the next START. If no START appears after STOP, no downtime is considered due to lack of information. B) As in 1, but only for full cycles (START->STOP) recorder in the provided data. Both cases are calculated and reported. These two interpretations help in case the data come in specific time intervals (hourly, daily, etc) and we may not be able to see the time a cycle started, but catch it while is on.
##

Q3:  Which production line had the most downtime and how much that was?

We used the total (definition A, above)  downtimes.

# 3) DATA WAREHOUSE SCHEMA AND TABLES

I designed the Data Warehouse (DWH) for the Witside Production Floor based on a Star Schema architecture. This design was chosen specifically for its strengths in handling event data and supporting analytical queries efficiently. The test dataset provided is simple, but this could become more complicated, with many additional production floor parameters, so this choice gives a good framework setup to build on in case of a scale up in the demands.

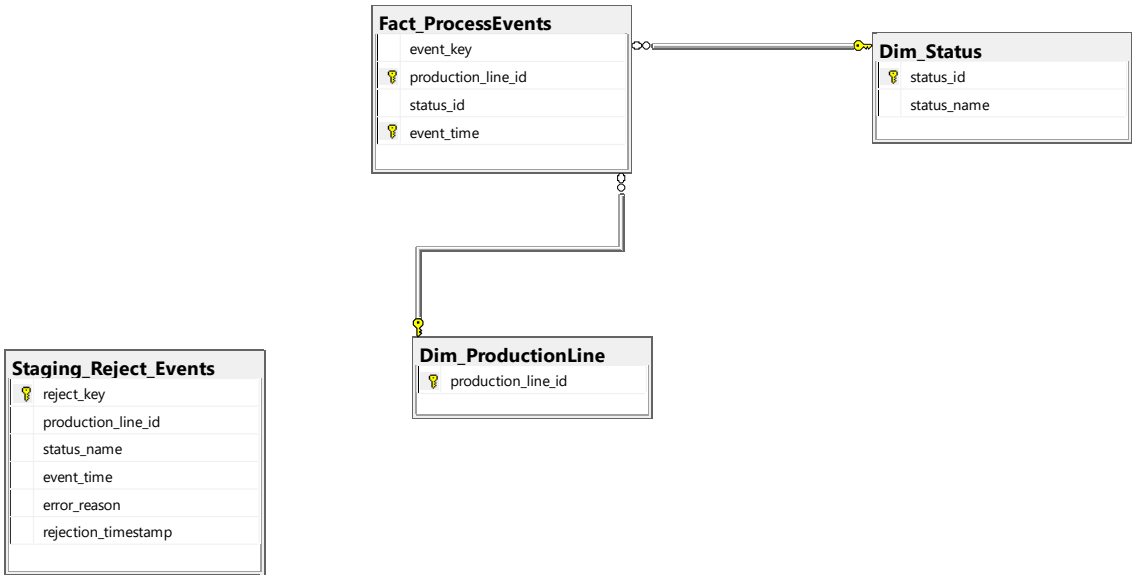


*Figure 1: Witside DWH (Witside_Production_Floor_DWH) schema*

The DWH comprises four tables: two Dimension Tables, one Fact Table, and one Staging (Data Quality) Table. Below I give an overview of the Tables contents.

| Table | Type | Purpose |
|---|---|---|
| Dim_Status | Dimension | Standardized lookup for event status names (START, ON, STOP). |
| Dim_ProductionLine | Dimension | List of all unique production line identifiers. |
| Fact_ProcessEvents | Fact | Core event records, measurement points, and foreign keys |
| Staging_Reject_Events | Staging | Collects quarantined data rows that violate data quality rules during ETL |

*Table 1: Summary of DWH Tables*

--TABLES DETAILS--

**Dim_Status**: Serves as a lookup dictionary, normalizing the event status names from the source data into a compact (TINYINT) integer identifier (status_id). This saves space in the large fact table and improves join performance.

**Dim_ProductionLine**: This table holds the natural key for the production lines, ensuring that all lines referenced in the fact table are valid and correctly cataloged. This dimension is designed to be loaded dynamically during the ETL process, inserting new production lines as they are encountered in the raw data.

**Fact_ProcessEvents**: This is the central measurement table, storing individual event records from the production floor. Fact_ProcessEvents uses a composite PK=(production_line_id, event_time). Note that event_time is the original timestamp which I renamed because timestamp is a reserved keyword, just to avoid possible confusion. The composite PK ensures that for any given production line, there is only one recorded event at a single point in time. This is essential for idempotency and prevents duplicate rows during data re-runs or incremental loads.

The event_key is defined as a UNIQUE NONCLUSTERED index. This allows it to serve as a high-performance, single-column identifier for all internal DWH queries (joins, updates) while keeping the original business composite key as the main clustered index for optimal data storage by line and time.

**Staging_Reject_Events**: This table is not part of the primary Star Schema but is used for maintaining data quality and providing auditability for ETL failures. The main motivation for this staging table is to quarantine rows that violate the crucial sequential process logic (e.g., measuring time in complete cycles requires a clear START -> ON -> STOP order). By isolating these rows, the Fact table remains clean for accurate uptime/downtime analysis.

# 4) ETL Architecture: Workload Distribution (Python vs. SQL):

The ETL (Extract, Transform, Load) pipeline is engineered as a Hybrid Architecture that leverages Python (Pandas and SQLAlchemy) for data handling and validation, and the SQL Database Engine for persistence, integrity, and complex analytics. This approach maximizes development speed and utilizes the database for tasks it performs best (set operations, aggregation). Here’s a breakdown of the division of tasks between the two tools:

**Data Extraction (E): Python (Pandas):** Normalizing the status strings (e.g., 'START') into their corresponding status_id integers (1) using the STATUS_MAP dictionary. This local transformation ensures the data is ready for the Star Schema without relying on complex database lookups mid-load.

**Data Cleaning/Normalization (T1): Python (Pandas)**: Applying the sequential logic check (current_status > previous_status) and quarantining invalid rows to Staging_Reject_Events. Python offers fine-grained control for row-by-row, logic-based validation and easy export of rejected data to a separate file for auditing.

**Data Validation/Quality Checks (T2): Python (Pandas):** Python queries the Dim_ProductionLine table to retrieve existing IDs, performs a Pandas set-difference operation to identify only the new IDs, and then uses SQL's bulk to_sql() method to insert the new dimensions. This minimizes database transactions.

**Dimension Management (T3): Hybrid (Python & SQL):** The clean, final Fact data is loaded directly into Fact_ProcessEvents using Pandas' to_sql(if_exists='append'). This delegates the actual, high-volume write operation to the database's optimized bulk loader, which is far faster than row-by-row inserts.

**Fact Loading (L): SQL (Bulk Insert via SQLAlchemy):** SQL is used to query the database and find the MAX(event_time) currently loaded. Python then uses this timestamp to filter the incoming Pandas DataFrame before bulk loading. This ensures idempotency and avoids unnecessary database comparisons during the load phase.

**Time-based Incremental Load (L): SQL & Python:** All complex time-series analysis (e.g., LEAD() window function to pair events and calculate time gaps, DATEDIFF(), and final SUM() aggregations) are handled entirely by dedicated SQL Views. This utilizes the database engine's core strength for parallel processing of set-based operations.

**Complex Aggregations (Analytics): SQL (Views):** All complex time-series analysis (e.g., LEAD() window function to pair events and calculate time gaps, DATEDIFF(), and final SUM() aggregations) are handled entirely by dedicated SQL Views. This utilizes the database engine's core strength for parallel processing of set-based operations.

**Final Report Generation: Python (Pandas & Report Writer):** Python uses pd.read_sql() to execute the queries based on the SQL Views, retrieves the results as DataFrames, and formats them into the final text report file.

!SQLAlchemy Note: The pipeline wants to offer flexibility in the SQL framework, so it is set to allow either use of SQL Server (T-SQL) or PostgreSQL. Note however that this first version is only completed and tested in T-SQL with SQL Server on Windows 11. PostgreSQL options currently exist as proof of concept and for future code expansion.

# 5) Pipeline Execution Flow:

The entire pipeline is executed via the main_runner.py script, which orchestrates four distinct phases in a sequential, idempotent manner:

1) DWH Setup (db_setup.py): Establishes the connection and performs necessary DDL (Data Definition Language). This phase creates the database if configured, and then drops/creates the required tables (dwh_sql_schema-tables.sql) and analytical views (dwh_sql_analytics-views.sql).

2) ETL Execution (etl_pipeline.py):

- Extracts data from the CSV source file.
- Transforms and cleans the data, performing status ID mapping and applying data quality checks.
- Loads new dimensions to Dim_ProductionLine.
- Loads new, clean event data to Fact_ProcessEvents incrementally.
- Outputs rejected data to the Staging_Reject_Events table and the quarantined_events.csv file.

3) Analytics and Reporting (dwh_analytics.py): Executes the three core business queries (Q1, Q2, Q3) by querying the pre-defined SQL views.

4) Final Report Generation: Compiles the execution log and the analytical results into the final analytics_report.txt file.

# 6) DATA WAREHOUSE VIEWS AND BUSINESS LOGIC:

The following SQL Views are created to help answering the business questions posed. The CREATE OR ALTER VIEW structure allows for easy, non-destructive updates to the business logic as requirements evolve. The current Views are used for business questions 2 and 3.

**View_Process_Cycle_Pairs**: Prepares the data to calculate the duration of full production cycles. It filters Fact_ProcessEvents for only START (1) and STOP (3) events. It uses the LEAD() window function, partitioned by production_line_id and ordered by event_time, to pull the details of the next relevant event onto the current event's row.

**View_Line_Process_Durations**: Calculates the clean run time for completed production processes (Start-to-Stop). This metric is reliable as it requires a specific, valid sequence. Filters View_Process_Cycle_Pairs to only include pairs where the current event was START (implied by the original filter) and the next event was a clean STOP (next_event_status_id = 3). Uses DATEDIFF(second, ...) to calculate the duration between the start and stop timestamps.

**View_Line_Event_Sequences**: Pairs every event in the Fact_ProcessEvents table with its immediate successor, irrespective of status. This is the foundation for calculating overall floor uptime and downtime. Uses LEAD() to get the next event_time and status_id for all events. Joins with Dim_Status to include the human-readable event_start_status. Calculates the gap_seconds between the current event and the next event.

**View_Total_Uptime_Downtime**: Aggregates the total time a line was available (Uptime) or unavailable (Downtime) based on the status reported at the beginning of the time interval (gap). Queries View_Line_Event_Sequences. Uses SUM() and CASE statements, grouped by production_line_id and a) sums gap_seconds where event_start_status was 'START' or 'ON' (for Uptime) and b) Sums gap_seconds where event_start_status was 'STOP' for downtime.

# 7) Key Analytical Queries

The dwh_analytics.py module executes the following SQL queries to produce the final analysis report. These queries are executed as pd.read_sql() statements against the DWH engine.

** Q1: Calculate All Process Cycle Durations for a Specific Line
Business Question: Show all completed process cycles and their durations for the line ID gr-np-47.

--Query--:

```
SELECT
    start_timestamp,
    stop_timestamp,
    duration_seconds
FROM
    View_Line_Process_Durations
WHERE
    production_line_id = ?
ORDER BY
    start_timestamp DESC;
```

** Q2: Calculate Total Floor Time (Uptime and Downtime) for All Lines
Business Question: Calculate the total aggregated uptime and downtime for all production lines.

--Query--

```
SELECT
    -- Aggregating all lines to provide floor-level totals
    SUM(total_uptime_seconds) AS total_floor_uptime_seconds,
    SUM(floor_downtime_seconds) AS total_floor_downtime_seconds,
    -- Added derived column for business clarity
    SUM(total_uptime_seconds + floor_downtime_seconds) AS
total_recorded_time_seconds
FROM
    View_Total_Uptime_Downtime;
```

** Q3: Rank Production Lines by Downtime
Business Question: Identify and rank the top 1 line(s) with the highest total floor downtime.

--Query--

```
SELECT TOP (?)
    production_line_id,
    floor_downtime_seconds
FROM
    View_Total_Uptime_Downtime
ORDER BY
    floor_downtime_seconds DESC;
-- The TOP value is parameterized by TOP_LINES_Q3 in config.py for T-SQL
compatibility
```

# Note On Future Architectural Considerations:

The current architecture is well-suited for the existing data volume and batch processing. Scaling the pipeline or shifting to real-time ingestion would require architectural adjustments and cloud tools, distributed computing frameworks, etc. E.g. if the data volume increases significantly (e.g., Petabytes of data), the primary bottleneck will shift from Python's memory to SQL's I/O and indexing, and therefore the architecture must shift to push more complex transformations and loads down to distributed processing frameworks or native database utilities.

# Analytics Results:

Here I present the results from the analysis of the given dataset.csv.

```
======================================================
=== ANALYTICS REPORTING ===
======================================================
```

--- Q1: Process Cycles for Line 'gr-np-47' ---

| start_timestamp | stop_timestamp | duration |
|:---|:---|:---|
| 2020-10-07 01:33:20 | 2020-10-07 02:03:20 | 30.0 minutes |
| 2020-10-07 02:15:02 | 2020-10-07 04:15:02 | 120.0 minutes |
| 2020-10-07 05:00:00 | 2020-10-07 05:55:17 | 55.28 minutes |

--- Q2: Total Floor Uptime and Downtime ---

| Total Up/Down-time | Total | In Full Cycles |
|:---|:---|:---|
| Total Uptime | 960.283 minutes | 205.283 minutes |
| Total Downtime | 56.667 minutes | 56.667 minutes |

--- Q3: Production Line with Most Downtime ---

| production_line_id | downtime |
|:---|:---|
| gr-np-47 | 56.6667 minutes |

The results show that for process there are three documented process cycles (START-> STOP) that lasted 20, 120 and 55.28 minutes, respectively.

The total floor (all processes) uptime was 205.28 minutes for the documented process cycles (therefore, only gr-np-47 has documented cycles in the data) and 960.283 minutes if we count all recorded uptime (START-> ON , ON-> ON, ON -> STOP).

Production line gr-np-47 is the only with documented STOP so it’s the process with the most downtime, with total downtime of 56.6667 minutes.