DSI310: Data Exploration and Preprocessing - Chapter 7: ETL - Landing Zone Data Management (Week 7)

1. The Gateway to the Data Platform: The Landing Zone Revisited

Welcome to Week 7 of DSI310. For the past six weeks, we have been meticulously building our understanding of the modern data ecosystem, brick by brick. We began by establishing the strategic role of a data engineer and the layered architecture of a data platform, from the raw **Landing Zone** to the polished **Analytics Zone**.[1] We then gained hands-on experience ingesting data from traditional relational databases like Chinook and Northwind, as well as modern sources like APIs and log files.[1] Most recently, we laid the intellectual groundwork for our unified data warehouse by exploring the theory and practice of dimensional modeling, designing our core dimension and fact tables, and understanding the critical distinction between OLTP and OLAP systems.[3]

This week, we return to the very first stage of the ETL (Extract, Transform, Load) pipeline and focus on the crucial process of managing data within the **Landing Zone**. This is not a trivial or simple step. The Landing Zone is the most foundational layer of a data lake, serving as the single, authoritative "source of truth" for all data that enters our system. The way we handle data here—by ensuring it is raw, immutable, and versioned—is what makes our entire data platform resilient, auditable, and ultimately trustworthy.[4]

The key concepts we will explore today are:

- The core principles that govern the Landing Zone, emphasizing immutability and the role of versioning.
- Best practices for developing robust, reusable, and modular Python scripts that can handle data ingestion from multiple sources.
- The critical importance of data quality assessment and validation at this early stage.
- Implementing basic error handling to create a resilient and self-healing data pipeline.

Our ultimate goal is to operationalize the "E" and "L" of ETL, creating an automated and reliable process that takes data from any source and lands it safely in its first home within our data lake. This process, while seemingly simple, is the foundation for creating the high-quality

data products and **data services** that our business relies on.[1]

2. The Core Principles of the Landing Zone

The Landing Zone is the most fundamental layer of a data lake, serving as the initial entry point for all data that enters the system.[1] Its design is based on a few core principles that ensure the long-term integrity and utility of the data platform.

- **Raw and Immutable Storage:** This is the most critical principle of the Landing Zone.[4] Data is ingested "as-is" from its source systems (e.g., relational databases, APIs, log files) and stored in its native format without any transformations or modifications. The rationale behind this is simple but profound: by saving an exact, raw copy of the original data, we create a "source of truth" that is always available. If a subsequent transformation in a downstream stage is found to be flawed, or if a new business requirement emerges, we can always reprocess the data from its raw state without needing to re-ingest it from the source, which might be impossible if the data is no longer available.[4] The only exception to this rule is the stripping of sensitive Personally Identifiable Information (PII) to comply with privacy regulations, replacing it with a unique, non-sensitive identifier.[5] This is a critical step for data governance and security.[5]
- **Versioning:** While we keep data in its raw form, it is crucial to version it.[1] This means we maintain a historical record of all changes to the source data. This allows for full auditability and the ability to travel back in time to analyze data from a specific point in the past. Versioning is often achieved by saving new data in a time-partitioned directory structure (e.g., data/landing_zone/raw/chinook/2024/09/20/) or by appending a timestamp to the filename itself.
- **Schema-on-Read:** A key advantage of the Landing Zone is its flexibility. Unlike a traditional data warehouse that enforces a strict schema at the time of loading (schema-on-write), a data lake allows for a more fluid approach called **schema-on-read**.[1] This means the schema is not imposed on the data until it is read and used by a downstream process. This flexibility is essential for handling unstructured and semi-structured data from sources like logs and APIs, where the schema might not be known in advance or might change over time.[1] The raw, immutable data in the Landing Zone is a testament to this principle, as it is ready for any future schema-on-read transformation that the business requires.

3. Building a Robust and Reusable Ingestion Engine with Python

To operationalize our data ingestion process, we need to move beyond simple, one-off scripts and develop robust, reusable Python code that can be scheduled to run automatically. This is a hallmark of professional data engineering. Our goal is to create a modular ingestion engine that can handle different data sources with a consistent and predictable process. We will continue to build on the code concepts introduced in Weeks 2 and 3.

3.1 A Modular and Reusable Code Structure

A well-structured data ingestion script should be composed of reusable functions, each with a single responsibility. This makes the code easier to read, maintain, and test. We can create separate functions for different data sources, such as ingest_from_sql, ingest_from_api, and ingest_from_log_files. A top-level orchestration script can then call these functions in a logical sequence.

3.2 Ingesting Data from Relational Databases (SQL)

Our script for ingesting data from the Chinook and Northwind databases must be robust enough to handle various tables and configurations. The following example demonstrates a reusable function for this purpose, building on our knowledge of SQLAlchemy and pandas.

Python

```python
import pandas as pd
from sqlalchemy import create_engine
import os

def ingest_from_sql(
    db_url: str,
    table_name: str,
    source_name: str,
    output_base_dir: str = "data/landing_zone/raw"
):
    """
    Ingests raw data from a SQL table and saves it to the Landing Zone.

    Args:
        db_url (str): The database connection URL.
        table_name (str): The name of the table to ingest.
        source_name (str): A name for the source system (e.g., 'chinook', 'northwind').
        output_base_dir (str): The base directory for the Landing Zone.
    """
    print(f"--- Starting ingestion for table: {table_name} from {source_name} ---")

    # Create the connection engine
    engine = create_engine(db_url)

    # Define the output directory path with time-based partitioning for versioning
    # This is a key best practice for a scalable data lake
    import datetime
    partition_date = datetime.date.today().strftime("%Y/%m/%d")
    output_dir = os.path.join(output_base_dir, source_name, table_name, partition_date)

    # Create the directory if it doesn't exist
    os.makedirs(output_dir, exist_ok=True)

    # Formulate the extraction query
    # We use SELECT * to get all columns and rows, adhering to the 'raw' principle
    sql_query = f"SELECT * FROM \"{table_name}\""

    try:
        # Use pandas to read the SQL query results directly into a DataFrame
        df = pd.read_sql(sql_query, engine)

        # Define the output file path. We'll use the Parquet format.
        # Parquet is a columnar format, which is excellent for storage efficiency
        # and later processing, and it preserves schema information.
        output_path = os.path.join(output_dir, f"{table_name}.parquet")

        # Save the DataFrame to a Parquet file
        df.to_parquet(output_path, index=False)

        print(f"Successfully ingested {len(df)} records into: {output_path}")
        print("--------------------------------------------------\n")

    except Exception as e:
        print(f"Error ingesting table {table_name}: {e}")
        raise # Re-raise the exception to be handled by a higher-level function

# Example of use:
# CHINOOK_URL = "sqlite:///path/to/Chinook_Sqlite.sqlite"
# NORTHWIND_URL = "sqlite:///path/to/northwind.db"
# ingest_from_sql(CHINOOK_URL, "Customer", "chinook")
# ingest_from_sql(NORTHWIND_URL, "Customers", "northwind")
```

This function is a single, reusable unit. It accepts all the necessary parameters, which makes it easy to integrate into a larger pipeline script that ingests multiple tables from both the Chinook and Northwind databases.

3.3 Ingesting Data from APIs and Log Files

Building on our work from Week 3, a robust ingestion script for APIs and log files would also be designed as a reusable function.

- **For APIs:** A function would take the API endpoint URL, any necessary authentication keys, and optional parameters as arguments. It would use the requests library to fetch the data and then convert the JSON response into a format like a pandas DataFrame before saving it as a Parquet file in the Landing Zone. The key here is to handle potential JSONDecodeError and various HTTP status codes gracefully.
- **For Log Files:** A function would take the log file path and a regular expression pattern as arguments. It would then read the file line by line to conserve memory, parse each line using the re module, and append the results to a list or a DataFrame. Once the entire file is processed, the resulting data would be saved to a Parquet file in the Landing Zone.

4. Data Quality Assessment at the Point of Ingestion

Data quality is a recurring theme throughout our course. While we will perform a deep dive into data cleansing in the coming weeks, it is crucial to perform basic data quality checks at the very first stage of ingestion. This is a form of **data validation**, where we check if the incoming data meets our basic expectations for integrity and consistency. Catching issues early saves a tremendous amount of time and effort in later stages of the pipeline.

What can we check for at the ingestion stage?

- **Schema Validation:** Does the data have the columns we expect? Are their data types correct? For our OmniCorp project, we would check if the Customers table from

Northwind has columns like CustomerID, CompanyName, and ContactName as expected.[2] If a column is missing, it's a sign of a potential issue with the source system that needs to be investigated.

- **Record Count Validation:** A simple but powerful check is to verify the number of records ingested. We can query the source system for a record count and compare it to the number of rows in the DataFrame after ingestion. A discrepancy here can signal a partial ingestion or an unexpected error.
- **Basic Data Sanity Checks:** We can perform quick checks on key fields to ensure they contain valid data.
  - **Null Value Checks:** For critical primary keys like CustomerID or ProductID, we can check for NULL values. The presence of NULL in these fields indicates a data quality issue at the source that must be addressed.
  - **Duplicate Record Checks:** We can check for duplicate records at the source using a combination of key fields. While duplicates can be handled later, identifying them at ingestion is an important form of **data profiling**.

The following Python code snippet shows how we can perform a simple data quality check on a DataFrame after ingestion, using the power of pandas.

Python

```python
# Assuming 'df' is the DataFrame after ingestion from a source
# Example: Data Quality Check on the 'Customers' table from Northwind

print("--- Performing basic data quality checks ---")

# 1. Check for expected columns
required_columns =
missing_columns = [col for col in required_columns if col not in df.columns]

if missing_columns:
    print(f"WARNING: Missing required columns: {missing_columns}. Investigation needed.")

# 2. Check for nulls in a critical column (CustomerID)
critical_key = 'CustomerID'
if df[critical_key].isnull().any():
    print(f"WARNING: Found {df[critical_key].isnull().sum()} NULL values in the {critical_key} column.")
    # In a real-world scenario, you might log this error and halt the pipeline.

# 3. Check for duplicates based on a key
```

```
duplicate_count = df.duplicated(subset=[critical_key]).sum()
if duplicate_count > 0:
    print(f"WARNING: Found {duplicate_count} duplicate records based on {critical_key}.")

print("--- Data quality checks complete ---")
```

5. Implementing Robust Error Handling and Resiliency

A production-grade data pipeline will inevitably face errors. These can range from a network outage that prevents a database connection to an unexpected change in an API's schema. A professional data engineer's job is not just to build a pipeline, but to build a **resilient** one that can handle these failures gracefully.

- **Using try...except blocks:** The most basic and important form of error handling is to wrap all critical operations in a try...except block. This allows the script to catch an error, log it, and either continue or fail in a controlled manner, preventing the entire pipeline from crashing. In our ingest_from_sql function, we included a try...except block to catch any exceptions during the data reading and saving process.
- **Graceful Failure and Alerting:** For severe errors (e.g., a connection failure to a critical source), the best approach is often to fail the job and send an alert to the data engineering team. It's better to have a failed job that can be manually investigated than to have a job that runs to completion but ingests incomplete or corrupted data.
- **Idempotency:** A key concept in designing a resilient pipeline is **idempotency**. An idempotent process is one that can be executed multiple times without changing the result beyond the initial execution. In the context of data ingestion, this means we can rerun our ingestion script for the same day without creating duplicate data in the Landing Zone. We can achieve this through our time-based directory structure: when the script runs again, it simply overwrites the existing files for that day, ensuring no duplicates are created.

6. The End Game: Building Data Products and Services

This week's lesson on Landing Zone data management is not an isolated task; it is the crucial first step in a much larger journey that culminates in the creation of valuable business assets. Our course syllabus highlights that the final output of our work should be **data products**, **data services**, and **machine-readable data**.[1]

- **Machine-Readable Data:** By ingesting data in a structured, binary format like Parquet and organizing it into a time-partitioned directory structure, we are already taking a massive step toward creating machine-readable data.[1] This data is not just for human consumption; it is optimized for high-performance analytical engines that can read and process it efficiently.
- **Data Products:** The Landing Zone is the source of all future data products. Our final FactSales table, designed from the raw data from Chinook and Northwind, is a prime example of a data product.[2] It is a curated, high-quality, and reliable asset that is ready to be consumed by other teams or services.
- **Data Services:** A data service is an application that provides access to data or insights derived from that data. A business intelligence dashboard that provides a real-time view of our sales metrics is a data service that is powered by our FactSales data product, which in turn is built from the raw data in our Landing Zone.[1]

Our journey through this course is a continuous loop of value creation. We start with raw, messy data, transform it into structured, clean data, and finally use that data to power business decisions. The robust Landing Zone we are building today is the critical first stage of this entire process.

7. Key Theories and Keywords

- **Landing Zone:** The foundational layer of a data lake, where raw, untransformed, and immutable data is stored from all sources. It serves as the single source of truth for the entire data platform.[1]
- **Immutability:** A core principle of the Landing Zone where data is never modified after it is first ingested. This ensures a complete historical record and allows for full reproducibility and auditability of the data pipeline.
- **Data Quality Assessment:** The process of evaluating data for correctness, completeness, consistency, and other quality dimensions. While full-scale cleansing happens in the Staging Zone, basic checks are performed at the ingestion stage.[1]
- **Data Validation:** The act of checking whether incoming data conforms to a set of predefined rules or expectations (e.g., checking for NULL values in a primary key column).
- **Error Handling:** The practice of anticipating and managing potential errors or failures in a data pipeline to ensure it remains resilient and reliable.
- **Idempotency:** A property of an operation that produces the same result regardless of how many times it is executed. Designing an idempotent ingestion process prevents data duplication and makes the pipeline more robust.
- **Data Product:** A ready-to-use, curated, and reliable dataset that is designed to serve a specific business purpose or use case. Our FactSales table is a data product.[1]

- **Data Service:** An application or system that provides access to data or insights derived from data products, such as a business intelligence dashboard or a real-time alerting system.
- **Machine-Readable Data:** Data that is formatted and structured in a way that can be easily processed and interpreted by computer systems, often through the use of formats like Parquet.[1]

Works cited

1. dsi310_แบบฟอร์มเค้าโครงการบรรยาย 2025
2. dsi310: Data Lake Modeling
3. OLTP vs. OLAP: Differences and Applications - Snowflake, accessed September 10, 2025, https://www.snowflake.com/en/fundamentals/olap-vs-oltp-the-differences/
4. ETL vs ELT: Key Differences, Comparisons, & Use Cases - Rivery, accessed September 10, 2025, https://rivery.io/blog/etl-vs-elt/
5. ETL vs ELT - Difference Between Data-Processing Approaches - AWS, accessed September 10, 2025, https://aws.amazon.com/compare/the-difference-between-etl-and-elt/