

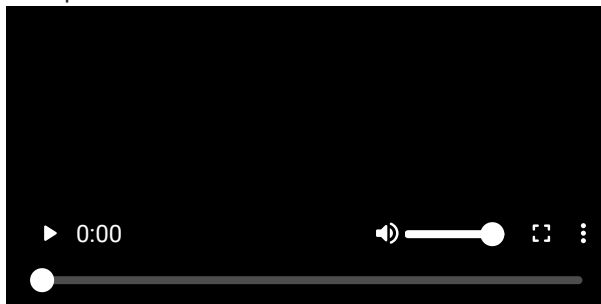
Cleanse, transform, and load data into Unity Catalog

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/1-introduction>

Introduction

Completed



Raw data rarely arrives in a format ready for analysis. Missing values, duplicate records, inconsistent data types, and poorly structured datasets are common challenges that data engineers must address before data can deliver business value. Without proper cleansing and transformation, analytics results become unreliable and downstream processes fail. **Unity Catalog** in Azure Databricks provides the governance foundation for managing transformed data assets throughout this process.

Azure Databricks offers a comprehensive toolkit for data quality and transformation work. **Data profiling** generates summary statistics that reveal issues like null percentages, unexpected value distributions, and data drift. **Type selection** ensures columns use appropriate data types for storage efficiency and query performance. **Duplicate and null handling** techniques let you identify and resolve common data quality problems using SQL or PySpark. **Transformation operations** including filtering, grouping, aggregation, joins, and set operators reshape data to meet analytical requirements.

Beyond basic transformations, you'll work with advanced reshaping techniques. **Denormalization** flattens related tables for faster query performance. **Pivoting** rotates row values into columns for cross-tabular analysis, while **unpivoting** reverses this process for normalized data structures. Finally, **loading strategies** like append, overwrite, and merge ensure transformed data lands correctly in

target tables—whether you're adding new records, replacing existing data, or synchronizing changes through upsert operations.

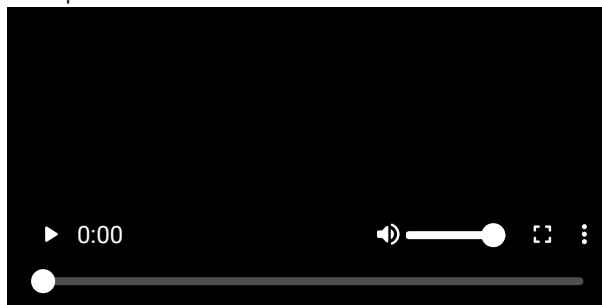
By the end of this module, you'll understand how to assess data quality through profiling, apply cleansing techniques to resolve common issues, transform data using SQL and PySpark operations, and load the results into Unity Catalog tables with the appropriate strategy for each scenario.

2. Profile data

<https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/2-profile-data-summary-statistics>

Profile data

Completed

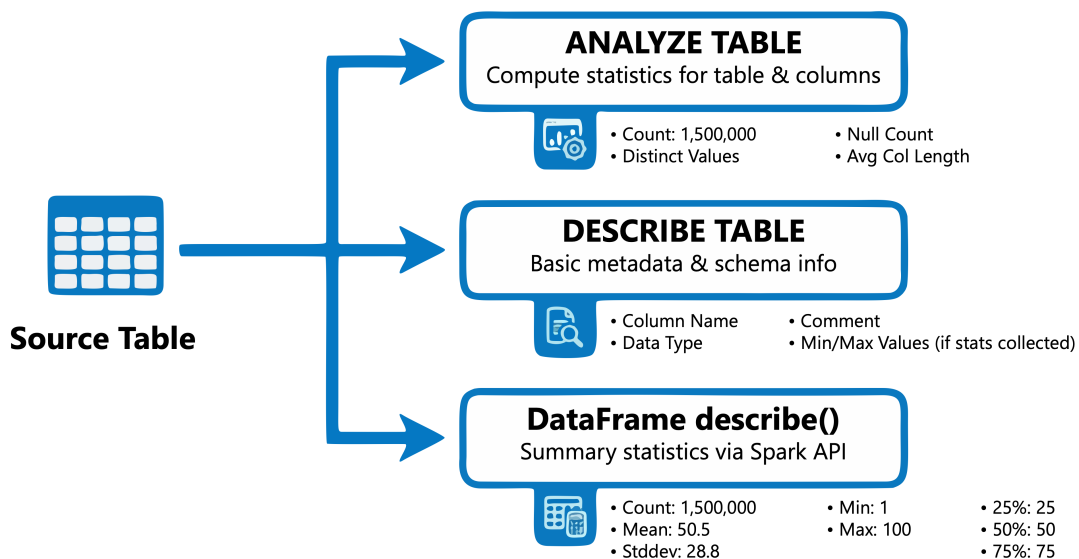


Before you cleanse or transform data, you need to understand what's in your tables. **Profiling data** means examining your datasets to generate summary statistics—counts, averages, value distributions, and null percentages—that reveal the quality and structure of your data. This foundation helps you identify issues like missing values, outliers, and unexpected patterns before they cause problems downstream.

In this unit, you learn how to generate summary statistics using SQL commands and the data profiling feature in Azure Databricks.

Generate summary statistics using SQL

Azure Databricks provides SQL commands that collect and display statistics about your tables. These statistics help the query optimizer choose efficient execution plans and give you insight into your data's characteristics.



Collect statistics with ANALYZE TABLE

The `ANALYZE TABLE` command computes statistics for a table or specific columns. These statistics include row counts, column-level metrics like minimum and maximum values, null counts, and distinct value counts.

```
-- Collect basic table statistics (row count and size)
ANALYZE TABLE sales.transactions COMPUTE STATISTICS;

-- Collect statistics for specific columns
ANALYZE TABLE sales.transactions COMPUTE STATISTICS FOR COLUMNS
  transaction_id, amount, customer_id;

-- Collect statistics for all columns
ANALYZE TABLE sales.transactions COMPUTE STATISTICS FOR ALL COLUMNS;
```

When you analyze a specific column, you get detailed metrics:

```
-- View column-level statistics
DESC EXTENDED sales.transactions amount;
```

This returns metrics including minimum value, maximum value, number of nulls, distinct count, average column length, and maximum column length.

View statistics with DESCRIBE TABLE

The `DESCRIBE TABLE EXTENDED` command displays collected statistics alongside table metadata:

```
DESCRIBE TABLE EXTENDED sales.transactions;
```

You can also use PySpark to retrieve table metadata and generate summary statistics:

```
# Generate summary statistics for numeric columns using the DataFrame API
df = spark.table("sales.transactions")
df.describe().show()
```

The `describe()` method returns count, mean, standard deviation, min, and max for numeric columns.

The output shows the table's size in bytes and row count, confirming that profiling has been performed. For Unity Catalog managed tables, predictive optimization automatically runs `ANALYZE` to keep statistics current.

Tip

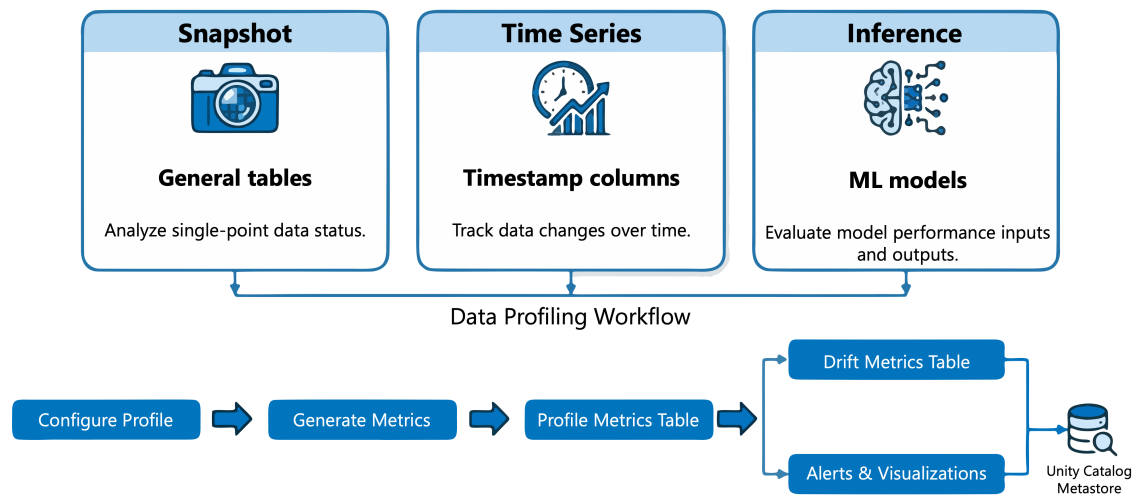
Run `ANALYZE TABLE` after bulk data loads or significant changes to ensure the query optimizer has accurate statistics for planning efficient queries.

Use data profiling in Unity Catalog

Beyond SQL commands, Azure Databricks provides **data quality monitoring** in Unity Catalog—an umbrella that covers two distinct capabilities:

- **Anomaly detection** monitors all tables in a schema automatically. It analyzes historical patterns to evaluate each table's freshness (how recently it was updated) and completeness (whether the expected number of rows arrived). You enable it once at the schema level in Catalog Explorer, and Azure Databricks handles the rest using intelligent scanning. Use anomaly detection for broad, automated coverage across all your tables without per-table configuration.
- **Data profiling** provides deep per-table statistical analysis—tracking distributions, null percentages, drift metrics, and model performance over time. You configure it individually for each table. Use data profiling for your most critical tables where you need detailed historical trends.

For data engineering work focused on cleansing and quality, you'll use both: anomaly detection as an early-warning system across all tables, and data profiling for in-depth monitoring of key tables.



Understand profile types

Data profiling supports three analysis types, each suited to different scenarios:

Profile type	Use case	Key characteristics
Snapshot	General-purpose tables	Computes metrics over the entire table with each refresh
Time series	Tables with timestamp columns	Tracks metrics across time windows to detect trends
Inference	ML model inference tables	Monitors model inputs, predictions, and accuracy over time

For data engineering tasks focused on data quality, the snapshot and time series profiles are most relevant.

Create a profile using the UI

To create a profile in Catalog Explorer:

1. Navigate to your table in Catalog Explorer.
2. Select the **Quality** tab.
3. If anomaly detection is **not** enabled for the schema, select **Enable**. If anomaly detection is already enabled, select **Configure**.
4. In the **Data Quality Monitoring** dialog, in the **Data profiling** field, select **Configure**.
5. Choose your profile type and configure options like granularity and scheduling.

The profile generates two metric tables: a profile metrics table with summary statistics and a drift metrics table tracking distribution changes over time.

Create a profile using the API

For programmatic access, use the Databricks SDK. Install or upgrade the SDK to get the current API:

```
%pip install "databricks-sdk>=0.68.0"
```

Then create a snapshot profile for a table:

```
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.dataquality import Monitor, DataProfilingConfig, Snapshot

w = WorkspaceClient()

# Retrieve the table ID and schema ID required by the API
table = w.tables.get(full_name="main.sales.transactions")
schema = w.schemas.get(full_name="main.sales")

w.data_quality.create_monitor(
    monitor=Monitor(
        object_type="table",
        object_id=table.table_id,
        data_profiling_config=DataProfilingConfig(
            output_schema_id=schema.schema_id,
            assets_dir="/Workspace/Users/user@example.com/monitoring",
            snapshot=SnapshotConfig()
        )
    )
)
```

To refresh the profile and update metrics:

```
from databricks.sdk.service.dataquality import Refresh

w.data_quality.create_refresh(
    object_type="table",
    object_id=table.table_id,
    refresh=Refresh(object_type="table", object_id=table.table_id)
)
```

Interpret profiling results

Profiling generates metrics that help you assess data quality and identify issues requiring attention.

Key metrics to examine

The profile metrics table contains statistics for each column:

Metric	What it reveals
<code>count</code> and <code>num_nulls</code>	Data completeness—high null percentages might indicate ingestion issues
<code>distinct_count</code>	Cardinality—unexpected values might suggest data quality problems
<code>min</code> , <code>max</code> , <code>avg</code> , <code>stddev</code>	Distribution characteristics—outliers or skewed distributions

Metric	What it reveals
frequent_items	Most common values—helps identify dominant categories or duplicates
quantiles	Value distribution—useful for detecting skewness

Detect data quality issues

When you examine profiling results, look for these common indicators:

- **High null percentages** in columns that should have values
- **Unexpected distinct counts**, such as unique identifiers that aren't unique
- **Value ranges outside expected bounds**, like negative amounts where only positives are valid
- **Distribution changes over time**, which can indicate data drift

The drift metrics table calculates statistical tests like the Kolmogorov-Smirnov (KS) test for numeric columns and chi-squared test for categorical columns. These tests quantify how much distributions have changed.

Understanding these patterns helps you design appropriate cleansing and validation rules for your data pipelines.

3. Choose column data types

<https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/3-choose-appropriate-column-data-types>

Choose column data types

Completed

Choosing the right data type for each column directly affects your data quality, storage efficiency, and query performance. When you select an inappropriate type, you risk losing precision, wasting storage space, or blocking operations that require specific type characteristics. In this unit, you learn how to evaluate your data and select the most appropriate column types in Azure Databricks.

Understand data type categories

Azure Databricks organizes data types into categories based on their characteristics and use cases. Understanding these categories helps you narrow down your options when choosing a type for a specific column.

Numeric types handle mathematical values and calculations. You have several options:

Type category	Types	Use case
Integral	TINYINT , SMALLINT , INT , BIGINT	Whole numbers of varying sizes
Decimal	DECIMAL(p, s)	Exact precision for financial data
Floating point	FLOAT , DOUBLE	Scientific calculations with approximation

Date-time types represent temporal data:

Type	Description
DATE	Calendar dates without time components
TIMESTAMP	Date and time with session timezone
TIMESTAMP_NTZ	Date and time without timezone

Text and binary types store character and byte sequences:

- STRING for variable-length text
- BINARY for raw byte sequences

Complex types organize nested or repeated data:

- ARRAY for ordered sequences of elements
- MAP for key-value pairs
- STRUCT for records with named fields

Select the right numeric type

Numeric data requires careful type selection because different types have different precision, range, and storage characteristics. Your choice affects both data integrity and performance.

For whole numbers, match the type to your value range:

Type	Range	Storage	Common use cases
TINYINT	-128 to 127	1 byte	Status codes, small counters
SMALLINT	-32,768 to 32,767	2 bytes	Year values, limited ranges
INT	-2.1 billion to 2.1 billion	4 bytes	Most identifiers, counts

Type	Range	Storage	Common use cases
BIGINT	-9.2 quintillion to 9.2 quintillion	8 bytes	Large IDs, row counts

When creating tables, specify the appropriate type for each column:

```
CREATE TABLE orders (
  order_id INT,
  status_code TINYINT,
  total_items SMALLINT,
  transaction_id BIGINT
);
```

For decimal numbers, the choice between **DECIMAL** and floating point types depends on your precision requirements:

```
-- DECIMAL(p,s): exact precision for financial calculations
-- p = total digits, s = digits after decimal point
CREATE TABLE transactions (
  transaction_id INT,
  amount DECIMAL(10,2),      -- Exact: up to 99,999,999.99
  tax_rate DECIMAL(5,4)      -- Exact: up to 9.9999
);

-- DOUBLE: approximate, suitable for scientific calculations
CREATE TABLE measurements (
  sensor_id INT,
  temperature DOUBLE,        -- Approximate, large range
  pressure DOUBLE
);
```

Important

Use **DECIMAL** for financial data where exact precision matters. Floating point types (**FLOAT** , **DOUBLE**) use binary representation and can introduce rounding errors with base-10 values like currency.

Choose temporal types correctly

Temporal data type selection depends on whether you need timezone awareness and what level of precision your application requires.

Use **DATE** when you only need calendar dates without time components:

```
CREATE TABLE employees (
  employee_id INT,
  hire_date DATE,
```

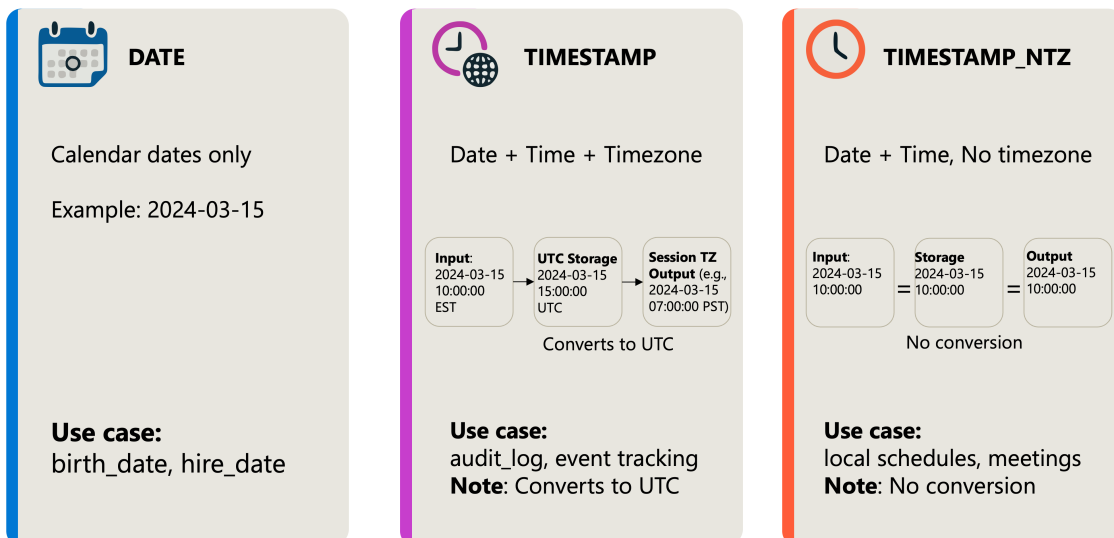
```
birth_date DATE
);
```

For timestamps, choose between timezone-aware and timezone-naive types:

```
-- TIMESTAMP: normalizes to UTC internally, applies session timezone on read
-- Use when tracking events across time zones
CREATE TABLE audit_log (
  log_id BIGINT,
  event_time TIMESTAMP,
  user_id INT
);

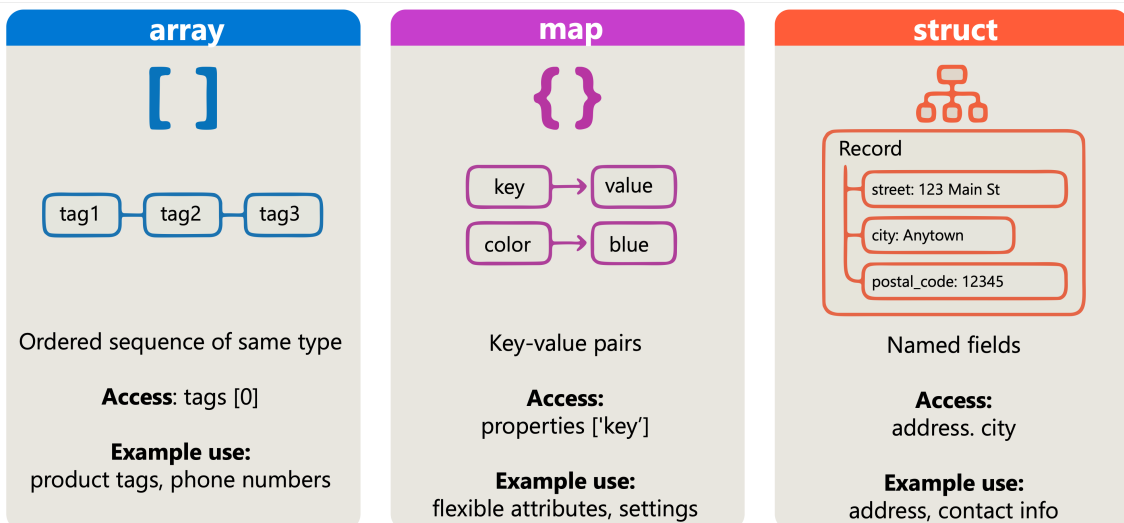
-- TIMESTAMP_NTZ: no timezone conversion
-- Use when the time should remain as entered
CREATE TABLE schedules (
  schedule_id INT,
  local_meeting_time TIMESTAMP_NTZ
);
```

The `TIMESTAMP` type converts values to UTC for storage and applies the session timezone when retrieving values. This behavior ensures consistent ordering across time zones but can cause confusion if you expect values to remain unchanged.



Work with complex types

Complex types let you store structured, nested, or repeated data within a single column. They're useful when your data doesn't fit a flat relational model.



Use `ARRAY` when you have multiple values of the same type:

```
-- Store a list of tags for each product
CREATE TABLE products (
  product_id INT,
  name STRING,
  tags ARRAY<STRING>
);

-- Query array elements
SELECT product_id, tags[0] AS primary_tag FROM products;
```

Use `MAP` for key-value pairs with dynamic keys:

```
-- Store flexible attributes as key-value pairs
CREATE TABLE devices (
  device_id INT,
  properties MAP<STRING, STRING>
);

-- Access map values by key
SELECT device_id, properties['firmware_version'] AS firmware FROM devices;
```

Use `STRUCT` when you have a fixed set of named fields:

```
-- Store address as a structured type
CREATE TABLE customers (
  customer_id INT,
  address STRUCT<street: STRING, city: STRING, postal_code: STRING>
);

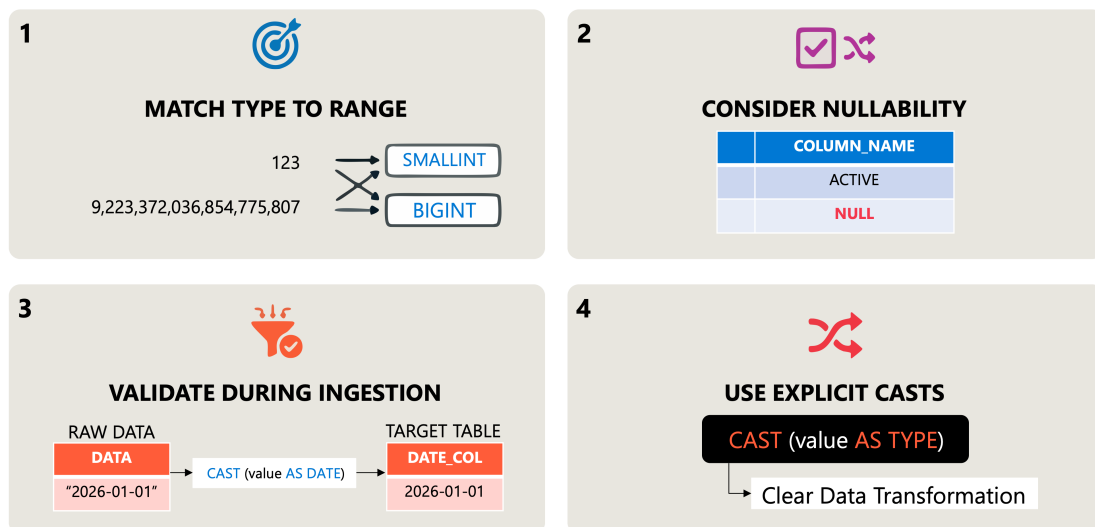
-- Access struct fields
SELECT customer_id, address.city FROM customers;
```

Tip

Complex types reduce the need for additional tables and joins, but they also make some operations more difficult. Consider your query patterns before choosing between normalized tables and complex types.

Apply type selection best practices

When evaluating data for type selection, consider these practical guidelines:



Match the type to the data's nature and range. Don't use `BIGINT` for values that never exceed a few thousand. The extra bytes multiply across millions of rows.

Consider nullability. If a column should never contain null values, document this constraint. While data types don't enforce nullability by default, you can add `NOT NULL` constraints to table definitions.

Validate types during ingestion. Cast incoming data to the target type early in your pipeline to catch type mismatches before they propagate:

```
-- Cast during data loading
INSERT INTO target_table
SELECT
    CAST(raw_id AS INT) AS id,
    CAST(raw_amount AS DECIMAL(10,2)) AS amount,
    CAST(raw_date AS DATE) AS transaction_date
FROM staging_table;
```

Use explicit casts when converting types. Implicit conversions can succeed silently even when data is lost. Explicit casts make your intentions clear and help identify problems during testing.

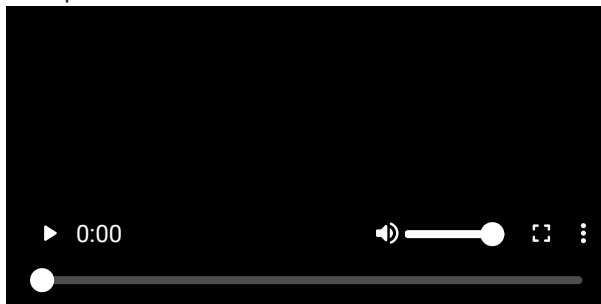
These practices help you build pipelines where data types serve as a form of documentation and validation, catching errors early rather than letting them affect downstream analytics.

4. Resolve duplicates and nulls

<https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/4-resolve-duplicate-missing-null-values>

Resolve duplicates and nulls

Completed



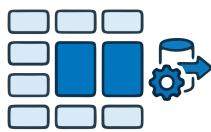
Data quality issues like duplicate records and null values can compromise your analytics and downstream processes. Whether duplicates arise from concurrent data loads or nulls appear due to incomplete source data, addressing these issues is a core responsibility for data engineers working in Azure Databricks.

In this unit, you learn how to identify and resolve duplicate, missing, and null values using both SQL and PySpark approaches.

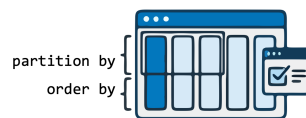
Identify duplicate records

Before you can remove duplicates, you need to find them. Duplicates typically occur when the same record appears multiple times in a table, often due to data integration issues, repeated ingestion, or concurrent write operations.

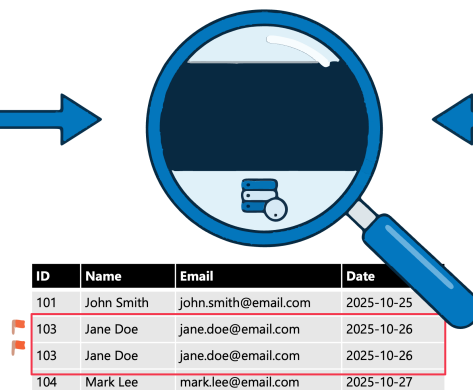
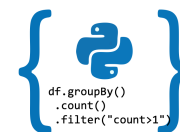
SQL GROUP BY



SQL QUALIFY



PySpark



Use GROUP BY and HAVING to find duplicates

The most straightforward way to identify duplicates is by grouping records and counting occurrences. The `HAVING` clause filters the grouped results to show only values that appear more than once.

```
SELECT
    customer_id,
    email,
    COUNT(*) AS occurrence_count
FROM sales.customers
GROUP BY customer_id, email
HAVING COUNT(*) > 1
ORDER BY occurrence_count DESC;
```

This query returns all combinations of `customer_id` and `email` that appear multiple times in the table, helping you understand the scope of duplication.

Use QUALIFY with window functions

The `QUALIFY` clause provides a more elegant approach by filtering the results of window functions directly, without requiring subqueries. This approach is particularly useful when you need to see the actual duplicate rows, not just counts.

```
SELECT *
FROM sales.customers
QUALIFY COUNT(*) OVER (PARTITION BY customer_id, email) > 1;
```

This query might return results like:

customer_id	email	name	created_at	status
1001	jane@contoso.com	Jane Doe	2024-01-15	Active
1001	jane@contoso.com	Jane D.	2024-03-20	Active
1002	bob@fabrikam.com	Bob Smith	2024-02-10	Pending
1002	bob@fabrikam.com	Robert Smith	2024-02-11	Active

With this pattern, you get all columns from the duplicate rows, making it easier to investigate the source of duplication.

Find duplicates using PySpark

In PySpark, you can use window functions combined with filtering to identify duplicate records:

```
from pyspark.sql.functions import count
from pyspark.sql.window import Window

window_spec = Window.partitionBy("customer_id", "email")

duplicates_df = (
    df.withColumn("row_count", count("*").over(window_spec))
      .filter("row_count > 1")
      .drop("row_count")
)

display(duplicates_df)
```

Alternatively, use the `exceptAll()` method to compare the original DataFrame against a deduplicated version:

```
df_duplicates = df.exceptAll(df.dropDuplicates(["customer_id", "email"]))
display(df_duplicates)
```

The `exceptAll()` method performs a set difference that preserves duplicates. It returns all rows from the first DataFrame that aren't in the second DataFrame, keeping duplicate occurrences. Here's how it works:

1. `df.dropDuplicates(["customer_id", "email"])` creates a DataFrame with one row per unique combination
2. `exceptAll()` compares the original DataFrame against this deduplicated version
3. For each unique combination, one occurrence is "matched" and removed, leaving only the extra duplicates

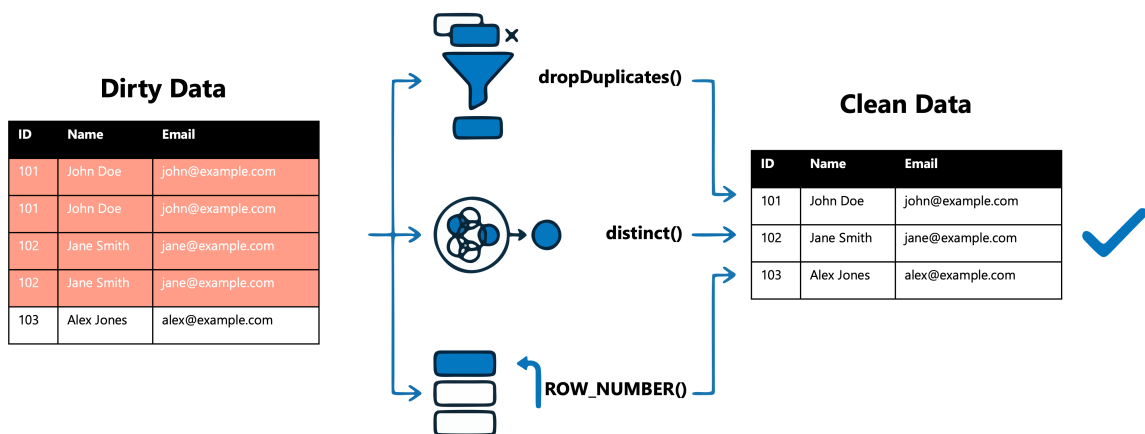
For example, if customer 1001 appears three times in the original DataFrame and once in the deduplicated version, `exceptAll()` returns two rows (the extra occurrences).

Note

Unlike `except()`, which removes all matching rows and returns distinct results, `exceptAll()` preserves the duplicate structure. If you have three identical rows in the original and one in the deduplicated set, `exceptAll()` returns exactly two rows.

Remove duplicate records

Once you've identified duplicates, you can choose a resolution strategy based on your data requirements.



Remove duplicates in PySpark

The `dropDuplicates()` method removes duplicate rows based on specified columns, keeping only the first occurrence:

```
df_clean = df.dropDuplicates(["customer_id", "email"])
```

For complete row deduplication across all columns, use `distinct()`:

```
df_unique = df.distinct()
```

Keep specific rows using ROW_NUMBER

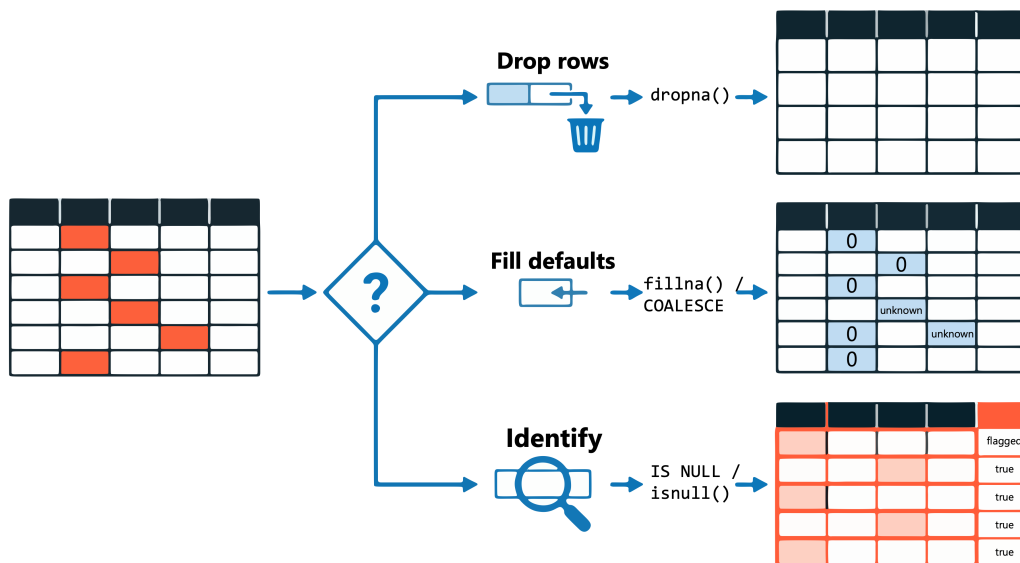
When you need control over which duplicate to keep, use `ROW_NUMBER()` with `QUALIFY` to select records based on criteria like the most recent update:

```
SELECT *
FROM sales.customers
QUALIFY ROW_NUMBER() OVER (
    PARTITION BY customer_id
    ORDER BY updated_at DESC
) = 1;
```

This query partitions records by `customer_id`, orders them by `updated_at` in descending order, and keeps only the most recently updated record for each customer.

Handle null and missing values

Null values represent missing or unknown data. Depending on your analysis requirements, you might need to remove rows with nulls, replace them with default values, or use statistical imputation.



Identify null values

Use the `isnull()` function or `IS NULL` operator to find records with missing values:

```
SELECT *
FROM sales.transactions
WHERE amount IS NULL
      OR customer_id IS NULL;
```

To count null occurrences per column:

```
SELECT
    COUNT(*) AS total_rows,
    COUNT(*) - COUNT(amount) AS null_amount_count,
    COUNT(*) - COUNT(customer_id) AS null_customer_count
FROM sales.transactions;
```

Drop rows with null values

In PySpark, the `dropna()` method removes rows containing null values:

```
# Drop rows where any column is null
df_clean = df.dropna()
```

```
# Drop rows where all columns are null
df_clean = df.dropna(how="all")

# Drop rows where specific columns are null
df_clean = df.dropna(subset=["customer_id", "amount"])
```

The `how` parameter controls the behavior: `"any"` drops rows with at least one null (default), while `"all"` drops rows only when all specified columns are null.

Fill null values with defaults

The `fillna()` method replaces null values with specified defaults:

```
# Fill all null values with a single value
df_filled = df.fillna(0)

# Fill specific columns with different values
df_filled = df.fillna({
    "amount": 0,
    "status": "Unknown",
    "quantity": 1
})
```

In SQL, use the `COALESCE` function to provide default values:

```
SELECT
    customer_id,
    COALESCE(amount, 0) AS amount,
    COALESCE(status, 'Unknown') AS status
FROM sales.transactions;
```

Choose the right strategy

The appropriate null handling strategy depends on your data context:

Scenario	Recommended approach
Nulls in required fields like identifiers	Drop the rows
Nulls in numeric fields for aggregation	Fill with zero or mean
Nulls in optional categorical fields	Fill with placeholder like "Unknown"
Nulls that would skew analysis	Drop or fill based on domain knowledge

Tip

Document your null handling decisions. Future analysts need to understand why certain values were replaced or removed to correctly interpret query results.

Understanding how to identify and resolve these data quality issues prepares you to build more reliable data pipelines. With clean data, your downstream transformations and analytics produce trustworthy results.

5. Transform data with filters and aggregations

<https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/5-transform-data-filter-group-aggregate>

Transform data with filters and aggregations

Completed



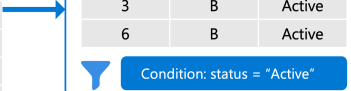
Data transformation turns raw data into formats suitable for analysis and reporting. **Filtering**, **grouping**, and **aggregating** are fundamental operations you use to shape data—selecting relevant records, organizing them into meaningful categories, and calculating summary statistics. These operations form the foundation of most data engineering pipelines.

In this unit, you learn how to filter rows based on conditions, group data by one or more columns, and apply aggregate functions to calculate summary values.

Filter data to select relevant rows

Filtering removes rows that don't meet your criteria, reducing dataset size and focusing on the data that matters. In Azure Databricks, you can filter data using PySpark's DataFrame API or SQL.

ID	Category	Value	Status
1	A	10	Active
2	A	10	Pending
3	B	15	Active
4	C	20	Pending
5	D	5	Inactive
6	B	20	Active



ID	Category	Status
1	A	Active
3	B	Active
6	B	Active

Condition: status = "Active"

Filter with PySpark

The `filter()` and `where()` methods work identically—use whichever you prefer for readability. The `col()` function references column names in your expressions.

```
from pyspark.sql.functions import col

# Filter customers with account balance greater than 1000
df_filtered = df_customer.filter(col("c_acctbal") > 1000)
display(df_filtered)
```

For multiple conditions, combine them with logical operators. Use `&` for AND and `|` for OR, wrapping each condition in parentheses:

```
# Filter customers in nation 20 with balance over 1000
df_filtered = df_customer.filter(
    (col("c_nationkey") == 20) & (col("c_acctbal") > 1000)
)

# Filter customers matching either of two IDs
df_filtered = df_customer.filter(
    (col("c_custkey") == 412446) | (col("c_custkey") == 412447)
)
```

Filter with SQL

SQL's `WHERE` clause provides the same filtering capability with familiar syntax:

```
-- Filter orders with status 'F' and total price above 50000
SELECT *
FROM orders
WHERE o_orderstatus = 'F'
      AND o_totalprice > 50000;
```

You can filter directly when reading from a table in PySpark using `spark.sql()`:

```
df_filtered = spark.sql("""
    SELECT *
    FROM sales.transactions
    WHERE amount > 100
          AND transaction_date >= '2024-01-01'
""")
```

Filter null values

Filtering null values requires special handling in Spark DataFrames. Use the `isNull()` and `isNotNull()` functions to identify or exclude null values:

```
# Filter rows where order_amount is not null
df_valid_orders = df.filter(col("order_amount").isNotNull())
```

```
# Filter rows where order_amount is null
df_null_orders = df.filter(col("order_amount").isNull())

# Alternative syntax using column object directly
df_valid_orders = df.filter(df.order_amount.isNotNull())
```

Important

Using Python's `None` with inequality operators like `!= None` doesn't reliably filter null values in Spark DataFrames. Null comparisons in SQL semantics don't evaluate to true or false—they return null. Always use `isNull()` or `isNotNull()` for correct null handling.

In SQL, use the `IS NULL` or `IS NOT NULL` operators:

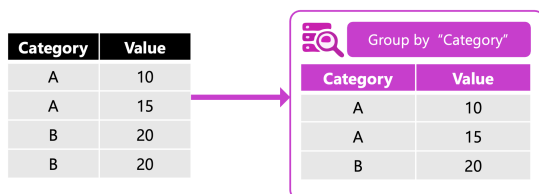
```
-- Filter orders with non-null amounts
SELECT *
FROM orders
WHERE order_amount IS NOT NULL;
```

For comprehensive null handling that removes entire rows containing null values, use the `dropna()` method covered in the unit on resolving duplicate and missing values:

```
# Remove rows where order_amount is null
df_clean = df.dropna(subset=["order_amount"])
```

Group data to organize records

Grouping organizes rows that share common values into categories. This prepares data for aggregation—once grouped, you can calculate statistics for each category.



Group with PySpark

Use the `groupBy()` method to specify one or more columns for grouping:

```
# Group by a single column
df_grouped = df_customer.groupBy("c_mktsegment")

# Group by multiple columns
df_grouped = df_customer.groupBy("c_mktsegment", "c_nationkey")
```

Grouping alone returns a `GroupedData` object. To see results, you must apply an aggregation.

Group with SQL

SQL uses the `GROUP BY` clause to organize rows:

```
-- Group sales by region
SELECT region, COUNT(*) as order_count
FROM sales
GROUP BY region;
```

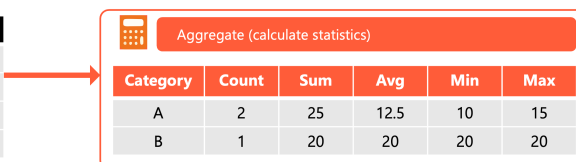
When grouping, every column in your `SELECT` statement must either appear in the `GROUP BY` clause or be wrapped in an aggregate function.

Aggregate data to calculate summary statistics

Aggregation calculates summary values—counts, sums, averages, and extremes—for each group. Common aggregate functions include:

Function	Description	Example use case
<code>count()</code>	Counts rows in each group	Number of orders per customer
<code>sum()</code>	Adds values in a column	Total revenue by product
<code>avg()</code>	Calculates the mean	Average order value by region
<code>min()</code>	Finds the smallest value	Earliest order date per customer
<code>max()</code>	Finds the largest value	Highest transaction amount

Category	Value
A	10
A	15
B	20
B	20



Aggregate (calculate statistics)					
Category	Count	Sum	Avg	Min	Max
A	2	25	12.5	10	15
B	1	20	20	20	20

Aggregate with PySpark

Use the `agg()` method to apply aggregate functions after grouping. Import functions from `pyspark.sql.functions`:

```
from pyspark.sql.functions import avg, sum, count, min, max

# Calculate average balance by market segment
df_segment_avg = df_customer.groupBy("c_mktsegment").agg(
    avg("c_acctbal").alias("avg_balance")
```

```
)  
display(df_segment_avg)
```

Apply multiple aggregations in a single statement:

```
# Multiple aggregations per group  
df_order_stats = df_order.groupBy("o_orderpriority").agg(  
    count("o_orderkey").alias("order_count"),  
    sum("o_totalprice").alias("total_revenue"),  
    avg("o_totalprice").alias("avg_order_value"),  
    max("o_totalprice").alias("max_order")  
)  
display(df_order_stats)
```

Aggregate with SQL

SQL aggregations work directly in `SELECT` statements:

```
-- Calculate statistics by dealership  
SELECT id,  
       COUNT(*) AS total_orders,  
       SUM(quantity) AS total_quantity,  
       AVG(quantity) AS avg_quantity,  
       MAX(quantity) AS max_quantity  
FROM dealer  
GROUP BY id  
ORDER BY total_quantity DESC;
```

Use `HAVING` to filter groups after aggregation:

```
-- Find high-volume dealers only  
SELECT id, SUM(quantity) AS total_quantity  
FROM dealer  
GROUP BY id  
HAVING SUM(quantity) > 20;
```

Chain transformations for readable pipelines

Spark's lazy evaluation lets you chain multiple transformations into a single, readable pipeline. The operations execute only when you call an action like `display()` or `show()`.

```
from pyspark.sql.functions import col, count  
  
# Chain filter, group, aggregate, and sort  
df_result = (  
    df_order  
    .filter(col("o_orderstatus") == "F")  
    .groupBy("o_orderpriority")  
    .agg(count("o_orderkey").alias("n_orders"))  
)
```

```
.sort(col("n_orders").desc())  
)  
display(df_result)
```

This approach improves readability and lets Spark optimize the entire operation sequence before execution. Each transformation builds on the previous one, creating a clear data flow from source to result.

Tip

When chaining transformations, place filters early in the pipeline to reduce the data volume that subsequent operations must process.

Understanding filtering, grouping, and aggregating prepares you to build efficient transformation pipelines. These operations combine naturally with joins and other transformations to create comprehensive data processing workflows.

6. Transform data with joins and set operators

<https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/6-transform-data-join-union-intersect-except>

Transform data with joins and set operators

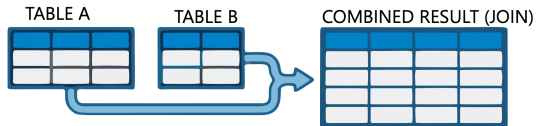
Completed

Data engineering workflows often require combining information from multiple datasets. **Joins** combine tables horizontally by matching rows on related columns, while **set operators** combine tables vertically by appending, intersecting, or excluding rows. Understanding when to use each approach helps you build efficient transformation pipelines.

In this unit, you learn how to join tables based on relationships and apply set operators to combine datasets with matching schemas.

Combine tables with joins

Joins merge rows from two tables based on a related column. The join type determines which rows appear in the result and how unmatched rows are handled.



Understand join types

Each join type serves a specific purpose in data transformation:



INNER JOIN



LEFT JOIN
(LEFT OUTER)



RIGHT JOIN
(RIGHT OUTER)



FULL JOIN
(FULL OUTER)



SEMI JOIN



ANTI JOIN



CROSS JOIN

Join type	Description	Use case
INNER	Returns only matching rows from both tables	Link orders with valid customers
LEFT	Returns all left table rows; adds NULLs for right table when no match	Include all customers, even without orders
RIGHT	Returns all right table rows; adds NULLs for left table when no match	Include all departments, even without employees
FULL	Returns all rows from both tables; adds NULLs for non-matches	Combine datasets to find gaps on either side
SEMI	Returns left table rows that have matches in the right table	Filter customers who placed orders
ANTI	Returns left table rows that have no match in the right table	Find customers who never placed orders
CROSS	Returns Cartesian product of all rows	Generate all possible combinations

Join tables with SQL

The `JOIN` clause combines tables based on a condition specified with `ON` or `USING`. Consider employee and department tables where you want to match employees to their departments:

```
-- Inner join: only employees with matching departments
SELECT e.id, e.name, e.deptno, d.deptname
FROM employee e
INNER JOIN department d ON e.deptno = d.deptno;
```

An inner join returns only three rows—employees whose department numbers exist in the department table. Employees in departments 4, 5, and 6 don't appear because those departments aren't in the department table.

When you need to include all employees regardless of department matches, use a left join:

```
-- Left join: all employees, NULL for missing departments
SELECT e.id, e.name, e.deptno, d.deptname
FROM employee e
LEFT JOIN department d ON e.deptno = d.deptno;
```

This returns all six employees. Those without matching departments show NULL in the `deptname` column.

Join DataFrames with PySpark

PySpark's `join()` method provides the same functionality. The `how` parameter specifies the join type:

```
df_employee = spark.table("employee")
df_department = spark.table("department")

# Inner join
df_inner = df_employee.join(
    df_department,
    on=df_employee.deptno == df_department.deptno,
    how="inner"
)
display(df_inner)
```

For left, right, or full outer joins, change the `how` parameter:

```
# Left join: keep all employees
df_left = df_employee.join(
    df_department,
    on="deptno",
    how="left"
)

# Full outer join: keep all rows from both tables
df_full = df_employee.join(
    df_department,
    on="deptno",
    how="full"
)
```

When the join column has the same name in both DataFrames, pass the column name as a string. This approach avoids duplicate columns in the result.

Filter data with semi and anti joins

Semi and anti joins are powerful for filtering data based on existence in another table. A semi join returns rows from the left table that have matches, without including columns from the right table:

```
-- Find employees who belong to listed departments
SELECT *
FROM employee
LEFT SEMI JOIN department ON employee.deptno = department.deptno;
```

An anti join returns the opposite—rows without matches:

```
-- Find employees in departments not listed
SELECT *
FROM employee
LEFT ANTI JOIN department ON employee.deptno = department.deptno;
```

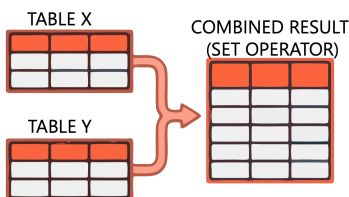
In PySpark, use `how="semi"` or `how="anti"`:

```
# Employees with matching departments
df_semi = df_employee.join(df_department, on="deptno", how="semi")

# Employees without matching departments
df_anti = df_employee.join(df_department, on="deptno", how="anti")
```

Combine rows with set operators

Set operators work vertically, combining rows from queries with the same column structure. Unlike joins, set operators don't match on conditions—they stack or compare entire result sets.



Understand set operators

Azure Databricks supports three set operators:

Operator	Description	Behavior
UNION	Combines rows from both queries	Removes duplicates by default; use <code>ALL</code> to keep them
INTERSECT	Returns rows that exist in both queries	Only matching rows appear
EXCEPT	Returns rows from the first query not in the second	Removes duplicates by default

Both queries must have the same number of columns with compatible data types.

Merge datasets with UNION

`UNION` appends rows from one query to another. By default, it removes duplicates:

```
-- Combine active and archived customers (no duplicates)
SELECT customer_id, name, email FROM active_customers
UNION
SELECT customer_id, name, email FROM archived_customers;
```

To preserve all rows including duplicates, use `UNION ALL`:

```
-- Combine all rows, keeping duplicates
SELECT customer_id, name, email FROM active_customers
UNION ALL
SELECT customer_id, name, email FROM archived_customers;
```

Use `UNION ALL` when you know there are no duplicates or when duplicates are meaningful. It performs better because it skips the deduplication step.

In PySpark, use `union()` for combining DataFrames. Note that PySpark's `union()` behaves like `UNION ALL`—it keeps duplicates:

```
df_active = spark.table("active_customers")
df_archived = spark.table("archived_customers")

# Combine both tables (keeps duplicates)
df_combined = df_active.union(df_archived)

# Remove duplicates if needed
df_distinct = df_active.union(df_archived).distinct()
```

Find common rows with INTERSECT

`INTERSECT` returns only rows that appear in both queries. This helps identify overlapping data:

```
-- Find customers in both active and archived tables
SELECT customer_id FROM active_customers
INTERSECT
SELECT customer_id FROM archived_customers;
```

This query identifies customers who somehow appear in both tables—useful for data quality checks or identifying records that need reconciliation.

Exclude rows with EXCEPT

`EXCEPT` returns rows from the first query that don't exist in the second. This helps identify gaps or unique records:

```
-- Find active customers not in the archived table
SELECT customer_id FROM active_customers
EXCEPT
SELECT customer_id FROM archived_customers;
```

You can also use `MINUS` as a synonym for `EXCEPT` :

```
-- Same result using MINUS syntax
SELECT customer_id FROM active_customers
MINUS
SELECT customer_id FROM archived_customers;
```

Like `UNION`, both `INTERSECT` and `EXCEPT` remove duplicates by default. Add `ALL` to preserve duplicates when needed.

Tip

When chaining set operations, remember that `INTERSECT` has higher precedence than `UNION` and `EXCEPT`. Use parentheses to control evaluation order.

Choose between joins and set operators

The choice between joins and set operators depends on your transformation goal. Joins combine columns from related tables horizontally, while set operators combine rows vertically.

Use **joins** when you need to:

- Enrich records by adding columns from related tables
- Match records based on key relationships
- Filter data based on existence in another table (semi/anti joins)

Use **set operators** when you need to:

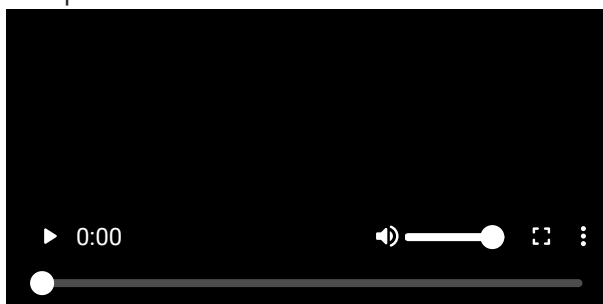
- Append records from multiple sources with identical schemas
- Find common records across datasets
- Identify records unique to one dataset

Both approaches form the foundation of effective data transformation pipelines. In the next unit, you explore how to load transformed data into target tables within Unity Catalog.

7. Transform data with denormalization and pivots

Transform data with denormalization and pivots

Completed



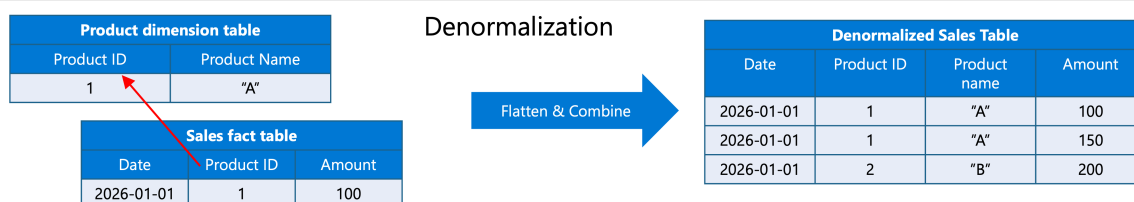
Reshaping data structure is a key part of building effective analytics solutions. **Denormalization**, **pivoting**, and **unpivoting** are transformation techniques that help you reorganize data between normalized and aggregated formats—each serving different analytical and modeling needs.

In this unit, you learn how to flatten normalized tables for faster queries, rotate rows into columns for cross-tabular analysis, and reverse that process when you need to normalize wide datasets.

Understand denormalization

Normalized databases reduce data redundancy by splitting information across multiple related tables. While this design works well for transactional systems, it creates challenges for analytics. Queries must join many tables, which slows performance and increases complexity.

Denormalization addresses these challenges by flattening related tables into fewer, wider tables. You combine data from dimension tables directly into fact tables, eliminating the need for joins at query time.



Consider a sales database with separate tables for orders, customers, and products. A normalized query requires multiple joins:

```
-- Normalized query with multiple joins
SELECT o.order_id,
       o.order_date,
       c.customer_name,
       c.region,
       p.product_name,
       p.category,
       o.quantity,
       o.total_price
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN products p ON o.product_id = p.product_id;
```

A denormalized table stores this information together:

```
-- Create a denormalized orders table
CREATE OR REPLACE TABLE sales.orders_denormalized AS
SELECT o.order_id,
       o.order_date,
       c.customer_name,
       c.region,
       p.product_name,
       p.category,
       o.quantity,
       o.total_price
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN products p ON o.product_id = p.product_id;
```

Queries against the denormalized table run faster because no joins are required. This approach is common when building star schemas and data marts for business intelligence tools. The tradeoff is increased storage and the need to refresh denormalized tables when source data changes.

Tip

Denormalize data in your gold layer tables when query performance matters more than storage efficiency. Keep normalized structures in bronze and silver layers for flexibility.

Pivot rows into columns

Pivoting transforms unique values from a column into separate columns, converting a "long" table format into a "wide" one. This transformation is particularly useful when preparing data for cross-tabular reports or dashboards.

Date	Category	Sales
2026-Q1	"Elec"	500
2026-Q1	"Cloth"	300
2026-Q2	"Elec"	600
2026-Q2	"Cloth"	400

Pivoting (Rows to Columns)

PIVOT (SUM(Sales))

Date	Elect	Cloth
2026-Q1	500	300
2026-Q2	600	400

The `PIVOT` clause requires an aggregate function because multiple source rows might map to the same cell in the pivoted result. You specify which column values become new column names using the `FOR` and `IN` clauses.

Consider quarterly sales data stored in a long format:

```
-- Create sample sales data
CREATE OR REPLACE TEMP VIEW quarterly_sales(year, quarter, region, revenue) AS
VALUES (2024, 1, 'East', 150000),
       (2024, 2, 'East', 175000),
       (2024, 3, 'East', 160000),
       (2024, 4, 'East', 200000),
       (2024, 1, 'West', 180000),
       (2024, 2, 'West', 165000),
       (2024, 3, 'West', 190000),
       (2024, 4, 'West', 210000),
       (2025, 1, 'East', 165000),
       (2025, 2, 'East', 185000);

-- View data in long format
SELECT * FROM quarterly_sales;
```

This query returns revenue as separate rows for each quarter. To compare quarters side by side, pivot the data:

```
-- Pivot quarters into columns
SELECT year, region, Q1, Q2, Q3, Q4
FROM quarterly_sales
PIVOT (
    SUM(revenue)
    FOR quarter IN (1 AS Q1, 2 AS Q2, 3 AS Q3, 4 AS Q4)
);
```

The result shows each region's quarterly revenue as columns, making year-over-year comparisons straightforward:

year	region	Q1	Q2	Q3	Q4
2024	East	150000	175000	160000	200000
2024	West	180000	165000	190000	210000
2025	East	165000	185000	null	null

You can apply multiple aggregate functions in a single pivot operation:

```
-- Pivot with multiple aggregations
SELECT year, Q1_total, Q1_avg, Q2_total, Q2_avg
FROM (SELECT year, quarter, revenue FROM quarterly_sales)
PIVOT (
    SUM(revenue) AS total,
    AVG(revenue) AS avg
    FOR quarter IN (1 AS Q1, 2 AS Q2)
);
```

This creates columns for both the sum and average of each quarter's revenue.

Unpivot columns into rows

Unpivoting performs the reverse transformation—it converts columns back into rows. Use unpivoting when you receive wide-format data that needs normalization for analysis or when preparing data for systems that expect a long format.

Date	Category	Sales
2026-Q1	"Elec"	500
2026-Q1	"Cloth"	300
2026-Q2	"Elec"	600
2026-Q2	"Cloth"	400

Pivoting (Rows to Columns)

PIVOT (SUM(Sales))

Date	Elect	Cloth
2026-Q1	500	300
2026-Q2	600	400

The `UNPIVOT` clause transforms specified columns into value-name pairs. You define a column to hold the values and another to hold the original column names.

```
-- Create wide-format data
CREATE OR REPLACE TEMP VIEW regional_targets(region, jan, feb, mar, apr) AS
VALUES ('North', 50000, 55000, 52000, 58000),
      ('South', 45000, 48000, 51000, 53000),
      ('East', 60000, 62000, 65000, 68000);

-- Unpivot monthly columns into rows
SELECT *
FROM regional_targets
UNPIVOT (
    target_amount FOR month IN (jan, feb, mar, apr)
);
```

The result converts each month column into rows:

region	month	target_amount
North	jan	50000
North	feb	55000
North	mar	52000
North	apr	58000

region	month	target_amount
South	jan	45000
...	...	...

By default, `UNPIVOT` excludes null values. To include them, specify `INCLUDE NULLS` :

```
-- Unpivot while keeping null values
SELECT *
FROM regional_targets
UNPIVOT INCLUDE NULLS (
    target_amount FOR month IN (jan, feb, mar, apr)
);
```

You can also assign custom aliases to the unpivoted column names:

```
-- Unpivot with custom month labels
SELECT *
FROM regional_targets
UNPIVOT (
    target_amount FOR month IN (
        jan AS `January`,
        feb AS `February`,
        mar AS `March`,
        apr AS `April`
    )
);
```

Note

The `UNPIVOT` clause requires Databricks Runtime 12.2 LTS or later.

Choose the right transformation

Each transformation serves specific analytical needs. Use denormalization when you need fast query performance for dashboards and reports. Choose pivot when analysts need side-by-side comparisons across categories. Apply unpivot when you need to normalize wide data for aggregation or machine learning pipelines.

These transformations often work together in data pipelines. You might unpivot source data to standardize its format, apply transformations, and then pivot results for final presentation—or denormalize the output into optimized tables for downstream consumers.

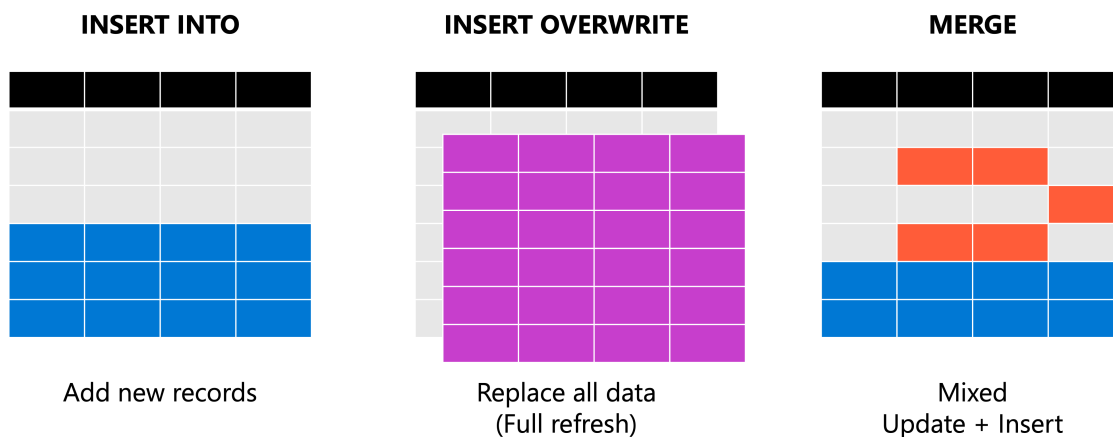
8. Load data with merge, insert, and append

Load data with merge, insert, and append

Completed



After you transform and cleanse data, the final step in your pipeline is loading it into target tables. Azure Databricks provides three core loading strategies: **append**, **overwrite**, and **merge**. Each strategy serves different use cases, and choosing the right one depends on how your source data relates to existing records.



In this unit, you learn how to append new rows with INSERT, replace data with overwrite operations, and synchronize changes using MERGE for upsert scenarios.

Append data with INSERT INTO

The most straightforward loading operation adds new rows to an existing table without modifying current data. Use INSERT INTO when you're loading fresh batches of data that don't overlap with existing records—such as daily transaction logs, new event streams, or incremental data extracts.

INSERT INTO works with both SQL and PySpark. The following examples show how to add rows to a sales table.

```
-- Append new records using VALUES
INSERT INTO sales.transactions (transaction_id, amount, transaction_date)
VALUES
    ('TXN001', 150.00, '2026-01-15'),
    ('TXN002', 275.50, '2026-01-15');

-- Append from a staging table or query
INSERT INTO sales.transactions
SELECT * FROM staging.daily_transactions
WHERE transaction_date = current_date();
```

In PySpark, you append data by writing a DataFrame with `append` mode:

```
# Append DataFrame to existing table
df_new_transactions.write.mode("append").saveAsTable("sales.transactions")

# Or use insertInto for existing tables
df_new_transactions.write.insertInto("sales.transactions")
```

Tip

When appending data, ensure your source data is deduplicated before loading. `INSERT INTO` doesn't check for duplicates—if you load the same batch twice, you get duplicate rows.

Replace data with `INSERT OVERWRITE`

Sometimes you need to replace existing data rather than add to it. `INSERT OVERWRITE` truncates the target table or partition before inserting new rows. This approach works well for reprocessing failed batches, refreshing lookup tables, or rebuilding aggregations.

```
-- Replace entire table contents
INSERT OVERWRITE sales.daily_summary
SELECT
    region,
    SUM(amount) as total_sales,
    COUNT(*) as transaction_count
FROM sales.transactions
WHERE transaction_date = current_date()
GROUP BY region;
```

For partitioned tables, you can overwrite specific partitions while leaving others intact:

```
-- Replace only January 2026 partition
INSERT OVERWRITE sales.monthly_report
PARTITION (report_month = '2026-01')
SELECT region, product_category, SUM(amount) as revenue
FROM sales.transactions
```

```
WHERE transaction_date BETWEEN '2026-01-01' AND '2026-01-31'  
GROUP BY region, product_category;
```

Delta Lake also supports selective overwrites with the `REPLACE WHERE` clause, which deletes rows matching a condition before inserting:

```
-- Replace transactions for a specific date range  
INSERT INTO sales.transactions  
REPLACE WHERE transaction_date BETWEEN '2026-01-01' AND '2026-01-07'  
SELECT * FROM staging.corrected_transactions;
```

In PySpark, use `overwrite` mode for full table replacement:

```
# Overwrite entire table  
df_refreshed.write.mode("overwrite").saveAsTable("sales.daily_summary")  
  
# Overwrite specific partition  
df_january.write.mode("overwrite").partitionBy("report_month").saveAsTable("sales.r
```

Important

Overwriting specific partitions by setting `partitionOverwriteMode=dynamic` is a legacy approach that only works on classic compute—it doesn't work on Databricks SQL warehouses or serverless compute. For new pipelines, use `REPLACE USING` instead.

Dynamically replace partitions with REPLACE USING

`REPLACE USING` is the recommended approach for dynamic partition overwrites. It works across all compute types—including Databricks SQL warehouses and serverless compute—and doesn't require Spark session configuration.

```
-- Replace only the partitions touched by the incoming data  
INSERT INTO sales.monthly_report  
  REPLACE USING (report_month)  
  SELECT region, product_category, SUM(amount) AS revenue, report_month  
  FROM staging.corrected_transactions  
  GROUP BY region, product_category, report_month;
```

The `REPLACE USING` clause atomically deletes rows in `sales.monthly_report` that match incoming rows on `report_month`, then inserts the new rows. Partitions not represented in the source query are left untouched.

When you need more flexible matching—such as replacing rows based on a custom condition or treating NULL values as equal—use `REPLACE ON` instead:

```
-- Replace rows matching a user-defined condition across source and target  
INSERT INTO sales.transactions
```

```
REPLACE ON (target.transaction_id = source.transaction_id)
SELECT * FROM staging.corrected_transactions AS source;
```

`REPLACE ON` requires Databricks Runtime 17.1 and above. Use `REPLACE WHERE` (covered earlier) when you need condition-based replacement without a source table reference.

Merge data for upsert operations

Real-world data pipelines often need to update existing records while inserting new ones. The `MERGE` statement handles this pattern—commonly called an "upsert"—by matching source rows to target rows and applying different actions based on whether a match exists.

`MERGE` is essential for slowly changing dimensions, synchronizing data from source systems, and processing change data capture (CDC) feeds.

```
MERGE INTO customers AS target
USING customer_updates AS source
ON target.customer_id = source.customer_id
WHEN MATCHED THEN
  UPDATE SET
    target.email = source.email,
    target.phone = source.phone,
    target.last_updated = current_timestamp()
WHEN NOT MATCHED THEN
  INSERT (customer_id, name, email, phone, created_date, last_updated)
  VALUES (source.customer_id, source.name, source.email, source.phone,
    current_timestamp(), current_timestamp());
```

This query updates email and phone for existing customers while inserting entirely new customer records. The `ON` clause defines the matching key—typically a primary key or business identifier.

Merge with conditional logic

Add conditions to `WHEN` clauses to handle more complex scenarios. For example, you might only update records that have actually changed:

```
MERGE INTO products AS target
USING product_feed AS source
ON target.sku = source.sku
WHEN MATCHED AND source.price <> target.price THEN
  UPDATE SET target.price = source.price, target.updated_at = current_timestamp()
WHEN MATCHED AND source.discontinued = true THEN
  DELETE
WHEN NOT MATCHED THEN
  INSERT *;
```

The `INSERT *` syntax inserts all columns from the source when they match the target schema.

Merge with PySpark

For programmatic control, use the Delta Lake Python API:

```
from delta.tables import DeltaTable

target_table = DeltaTable.forName(spark, "customers")

target_table.alias("target").merge(
    source_df.alias("source"),
    "target.customer_id = source.customer_id"
).whenMatchedUpdate(set={
    "email": "source.email",
    "phone": "source.phone",
    "last_updated": "current_timestamp()"
}).whenNotMatchedInsert(values={
    "customer_id": "source.customer_id",
    "name": "source.name",
    "email": "source.email",
    "phone": "source.phone",
    "created_date": "current_timestamp()",
    "last_updated": "current_timestamp()"
}).execute()
```

For simpler cases where source and target schemas match, use the convenience methods:

```
target_table.alias("target").merge(
    source_df.alias("source"),
    "target.customer_id = source.customer_id"
).whenMatchedUpdateAll(
).whenNotMatchedInsertAll(
).execute()
```

Choose the right loading strategy

Your choice of loading operation depends on the relationship between source and target data:

Scenario	Operation	When to use
New records only	INSERT INTO / append	Daily logs, event streams, incremental extracts
Full refresh	INSERT OVERWRITE	Lookup tables, aggregations, failed batch reprocessing
Mixed updates and inserts	MERGE	CDC feeds, dimension tables, data synchronization
Selective replacement by predicate	REPLACE WHERE	Correcting specific date ranges or logical conditions
Dynamic partition replacement (all compute types)	REPLACE USING	Recommended for partition overwrites on SQL warehouses and serverless

Scenario	Operation	When to use
Custom-condition row replacement	REPLACE ON	Partition overwrites with NULL-equality or cross-column conditions

Important

MERGE operations require a unique match between source and target rows. If multiple source rows match the same target row, the operation fails. Deduplicate your source data before merging.

Understanding these loading patterns prepares you to build reliable data pipelines that maintain data integrity while efficiently processing both new and changed records.

9. Exercise - Cleanse, Transform, and Load Data into Unity Catalog

<https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/9-exercise-cleanse-transform-load-data-into-unity-catalog>

Exercise - Cleanse, Transform, and Load Data into Unity Catalog

Completed

Now it's your chance to cleanse, transform, and load data into Unity Catalog. In this lab, you clean and reshape raw real estate data in Azure Databricks. You choose the right data types for prices and timestamps, remove duplicate listings and fill missing values using PySpark, and combine data across tables with inner and left joins. You also use SQL PIVOT and UNPIVOT to restructure market statistics for trend analysis.

Note

To complete this lab, you need an [Azure subscription](#) in which you have administrative access.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

10. Module assessment

<https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/10-knowledge-check>

Module assessment

Completed

11. Summary

<https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/11-summary>

Summary

Completed

Throughout this module, you explored the complete data engineering workflow for cleansing, transforming, and loading data into **Unity Catalog** tables in Azure Databricks. From initial data profiling to final loading operations, each technique builds on the previous to deliver high-quality, well-structured data ready for analysis.

You learned how **data profiling** using `ANALYZE TABLE` and Unity Catalog's monitoring features reveals data quality issues like null percentages, duplicate records, and distribution anomalies. You explored how selecting **appropriate data types** affects storage efficiency, query performance, and data integrity—particularly the importance of using `DECIMAL` for financial calculations. You practiced identifying and resolving **duplicates and null values** using techniques like `QUALIFY` with window functions, `dropDuplicates()`, and `fillna()` methods.

For data transformation, you applied **filtering, grouping, and aggregation** operations to shape data for analytical requirements. You combined datasets using **joins** for horizontal merging and **set operators** like `UNION`, `INTERSECT`, and `EXCEPT` for vertical combinations. You reshaped data structures through **denormalization** for query performance, **pivoting** for cross-tabular analysis, and **unpivoting** to normalize wide datasets.

Finally, you implemented **loading strategies** that match your data scenarios: `INSERT INTO` for appending new records, `INSERT OVERWRITE` for replacing data, and `MERGE INTO` for upsert operations that synchronize changes. As you build data pipelines, apply these techniques

systematically—profile first to understand your data, cleanse to resolve quality issues, transform to meet analytical requirements, and load with the appropriate strategy to maintain data integrity in your lakehouse.