

Implement and manage data quality constraints with Azure Databricks

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/1-introduction>

Introduction

Completed



Data quality issues can derail analytics projects, corrupt business reports, and erode confidence in your data platform. When invalid data enters your tables—whether through type mismatches, missing required values, or unexpected schema changes—problems compound as that data flows through pipelines and reaches downstream consumers. Implementing robust **data quality constraints** at the point of ingestion creates a foundation of trust in your data assets.

Azure Databricks provides multiple mechanisms for enforcing data quality within **Unity Catalog**. **Schema enforcement** validates data types automatically when writing to Delta Lake tables. **Table constraints** define rules that reject invalid records at write time. **Pipeline expectations** in Lakeflow Spark Declarative Pipelines enable real-time quality checks on streaming data with configurable actions for violations. Together, these capabilities form a comprehensive approach to maintaining data integrity.

Throughout this module, you explore practical techniques for implementing data quality constraints. You learn how to enforce data type checks using **schema validation**, **explicit casting**, and **CHECK constraints**. You discover strategies for managing **schema drift** when source systems evolve over time. You implement validation checks for **nullability**, **uniqueness**, and **value ranges**. Finally, you

master pipeline expectations that monitor data quality metrics and take automated actions when violations occur.

By combining these approaches, you build pipelines that catch quality issues early, prevent invalid data from reaching production tables, and provide visibility into the health of your data. These skills are essential for data engineers working with Unity Catalog who need to maintain high data quality standards across their organization's data estate.

2. Implement validation checks

<https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/2-implement-validation-checks>

Implement validation checks

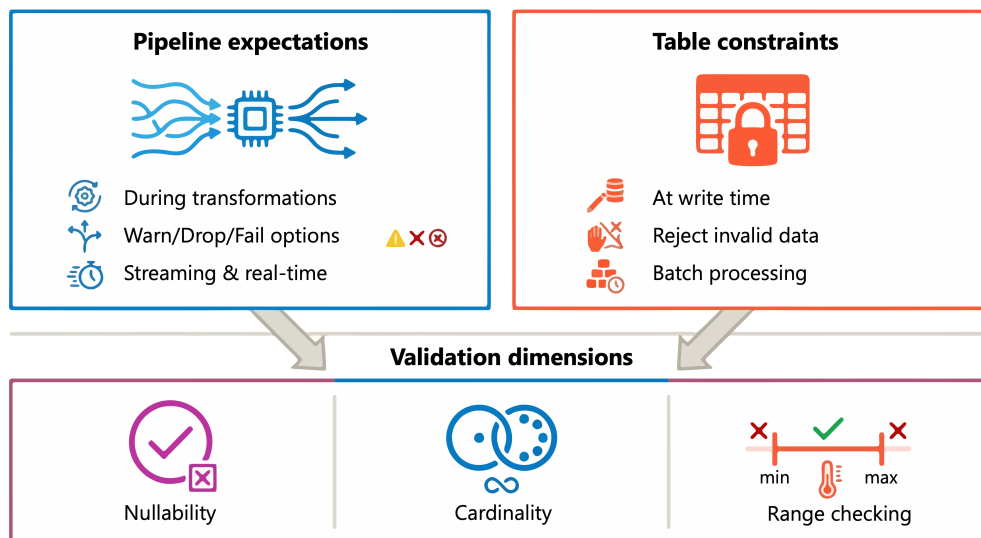
Completed

Data pipelines often encounter records with missing values, duplicate identifiers, or values that fall outside acceptable ranges. Without validation checks in place, these data quality issues propagate downstream, causing incorrect analytics, failed reports, and unreliable business decisions. Validation checks catch these problems at the point of data ingestion, ensuring that only quality data flows through your pipeline.

In this unit, you learn how to implement validation checks for nullability, data cardinality, and range checking using Lakeflow Spark Declarative Pipelines expectations and Delta Lake table constraints.

Understand validation approaches in Azure Databricks

Azure Databricks provides two primary mechanisms for implementing validation checks: pipeline expectations and table constraints. Each approach serves different scenarios and offers distinct capabilities.



Pipeline expectations apply validation during data transformations in Lakeflow Spark Declarative Pipelines. Expectations let you warn, drop invalid records, or fail the pipeline when data violates your rules. This approach works well for streaming tables and materialized views where you need real-time quality control.

Table constraints enforce rules directly on Delta Lake tables. Constraints reject invalid data at write time, preventing bad records from ever entering your tables. This approach suits batch processing and scenarios requiring strict data integrity guarantees.

With these approaches, you can validate three key dimensions of data quality: nullability (ensuring required values exist), data cardinality (verifying uniqueness where expected), and range checking (confirming values fall within acceptable bounds).

Implement nullability checks

Nullability validation ensures that required columns contain values. Consider a customer table where `email` and `customer_id` must always have values. You can implement these checks using expectations in your pipeline.

Using Python, add expectation decorators to your streaming table definition:

```
from pyspark import pipelines as dp

@dp.table()
@dp.expect_or_drop("valid_email", "email IS NOT NULL")
@dp.expect_or_drop("valid_customer_id", "customer_id IS NOT NULL")
def customers():
    return spark.readStream.table("raw.customers")
```

The same validation in SQL uses the `CONSTRAINT` clause:

```
CREATE OR REFRESH STREAMING TABLE customers(
  CONSTRAINT valid_email EXPECT (email IS NOT NULL) ON VIOLATION DROP ROW,
  CONSTRAINT valid_customer_id EXPECT (customer_id IS NOT NULL) ON VIOLATION DROP ROW
) AS SELECT * FROM STREAM(raw.customers);
```

For Delta Lake tables outside of pipelines, use `NOT NULL` constraints in the table definition:

```
CREATE TABLE customers (
  customer_id INT NOT NULL,
  email STRING NOT NULL,
  first_name STRING,
  last_name STRING
);
```

You can also add nullability constraints to existing tables:

```
ALTER TABLE customers ALTER COLUMN email SET NOT NULL;
```

Note

Before adding a `NOT NULL` constraint to an existing table, Azure Databricks verifies that all existing rows satisfy the constraint. The operation fails if any null values exist in the column.

Validate data cardinality

Cardinality validation ensures that columns expected to contain unique values actually do. This check is essential for primary keys, transaction identifiers, and other fields where duplicates indicate data quality problems.

Pipeline expectations can validate cardinality by checking for conditions that indicate uniqueness issues. For example, you can verify that a Social Security Number appears only once per person:

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

@dp.table()
@dp.expect("unique_ssn_per_person", "ssn_count = 1")
def employees():
    df = spark.table("raw.employees")
    w = Window.partitionBy("ssn")
    return df.withColumn("ssn_count", count("*").over(w))
```

For more comprehensive cardinality checks, combine expectations with aggregation logic in your transformation:

```
CREATE OR REFRESH MATERIALIZED VIEW order_validation AS
SELECT
    order_id,
    COUNT(*) as occurrence_count
FROM orders
GROUP BY order_id
HAVING COUNT(*) > 1;
```

This materialized view identifies any duplicate order IDs, enabling you to investigate and resolve cardinality issues.

Delta Lake supports primary key constraints that document expected uniqueness, though these constraints are informational and not enforced:

```
CREATE TABLE orders (
    order_id INT NOT NULL,
    customer_id INT,
    order_date DATE,
    CONSTRAINT orders_pk PRIMARY KEY (order_id)
);
```

Tip

While primary key constraints aren't enforced, they help query optimization and document your data model's intended structure. Use pipeline expectations to actively enforce uniqueness.

Apply range checking

Range validation confirms that numeric, date, and other values fall within acceptable bounds. This check catches data entry errors, system glitches, and integration issues that produce out-of-range values.

Define range expectations using comparison operators or the `BETWEEN` clause:

```
@dp.table()
@dp.expect_or_fail("valid_age", "age BETWEEN 0 AND 150")
@dp.expect_or_fail("valid_salary", "salary >= 0")
@dp.expect_or_fail("valid_hire_date", "hire_date <= current_date()")
def employees():
    return spark.readStream.table("raw.employees")
```

In SQL, apply the same range checks:

```
CREATE OR REFRESH STREAMING TABLE transactions(
    CONSTRAINT valid_amount EXPECT (amount > 0) ON VIOLATION DROP ROW,
    CONSTRAINT valid_date EXPECT (transaction_date <= current_date()) ON VIOLATION
```

```
CONSTRAINT valid_quantity EXPECT (quantity BETWEEN 1 AND 10000) ON VIOLATION DO  
) AS SELECT * FROM STREAM(raw.transactions);
```

For Delta Lake tables, use `CHECK` constraints to enforce ranges at write time:

```
ALTER TABLE employees ADD CONSTRAINT valid_age CHECK (age >= 18 AND age <= 120);  
ALTER TABLE transactions ADD CONSTRAINT positive_amount CHECK (amount > 0);
```

Range checks can also validate business rules that combine multiple conditions:

```
@dp.expect("valid_discount", ""  
    discount_percent >= 0  
    AND discount_percent <= 100  
    AND (discount_percent <= 50 OR customer_tier = 'PREMIUM')  
    """)
```

This expectation ensures discounts stay within bounds and enforces that only premium customers receive discounts over 50%.

Choose actions for validation failures

When validation fails, you choose how your pipeline responds. Each action suits different business requirements and data criticality levels.

Action	Use case	Behavior
Warn (default)	Monitoring and analysis	Invalid records write to target; metrics logged
Drop	Data cleansing	Invalid records removed before write
Fail	Critical data integrity	Pipeline stops; transaction rolls back

Use the **warn** action when you need visibility into data quality issues without blocking data flow:

```
@dp.expect("has_phone", "phone_number IS NOT NULL")
```

Use **drop** when invalid records should be filtered out silently:

```
@dp.expect_or_drop("complete_address", "street IS NOT NULL AND city IS NOT NULL")
```

Use **fail** when data integrity is critical and invalid records are unacceptable:

```
@dp.expect_or_fail("valid_account_balance", "balance >= 0")
```

You can view expectation metrics in the pipeline UI by selecting a dataset with expectations and opening the **Data quality** tab. These metrics help you monitor validation pass rates and identify systematic data quality issues.

Now that you understand how to implement validation checks, you can ensure your data pipelines maintain quality standards for nullability, cardinality, and value ranges.

3. Implement data type checks

<https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/3-implement-data-type-checks>

Implement data type checks

Completed



Every column in your dataset has an expected data type, and when incoming data doesn't match that expectation, problems cascade downstream. A string value where an integer belongs can break calculations, corrupt aggregations, or cause pipeline failures. Data type checks ensure that each field contains values of the correct type before they enter your tables.

In this unit, you learn how to implement data type checks using schema enforcement, explicit type casting, and validation constraints in Azure Databricks.

Understand schema enforcement

Delta Lake validates data types automatically when you write data to a table. This built-in mechanism, called **schema enforcement**, compares the data types of incoming columns against the target table's schema. When a type mismatch occurs, Delta Lake attempts to safely cast the value. If the cast fails, the write operation raises an error.

Schema enforcement applies the following rules during insert operations:

- All inserted columns must exist in the target table
- All column data types must match or be safely castable to the target types

Consider a table with an integer column:

```
CREATE TABLE inventory (  
  product_id INT,  
  quantity INT,  
  last_updated DATE  
);
```

When you insert data where `quantity` is a string that represents a number, Delta Lake casts it to the target type:

```
-- This succeeds because '100' can be cast to INT  
INSERT INTO inventory VALUES (1, '100', '2026-01-15');
```

However, inserting a string that can't be converted to an integer causes the operation to fail:

```
-- This fails because 'fifty' cannot be cast to INT  
INSERT INTO inventory VALUES (2, 'fifty', '2026-01-15');
```

Schema enforcement provides a first line of defense against type mismatches. For more control over how mismatches are handled, you can use explicit casting.

Use explicit type casting

The `cast` function converts values from one data type to another. When the conversion fails, an error is raised. The `try_cast` function works the same way but returns `NULL` instead of raising an error when the conversion fails.

Use `cast` when you want strict validation that stops processing on invalid data:

```
SELECT  
  cast(raw_amount AS DECIMAL(10,2)) AS amount,  
  cast(raw_date AS DATE) AS transaction_date  
FROM staging_data;
```

Use `try_cast` when you want to identify invalid values without failing the query:

```
SELECT  
  raw_amount,  
  try_cast(raw_amount AS DECIMAL(10,2)) AS validated_amount,  
  CASE  
    WHEN try_cast(raw_amount AS DECIMAL(10,2)) IS NULL  
    THEN 'Invalid amount format'  
    ELSE 'Valid'  
  END AS validation_status  
FROM staging_data;
```

With `try_cast`, records with invalid values return `NULL` for the cast column. You can then filter, flag, or quarantine these records based on your data quality requirements.

Implement type validation with constraints

CHECK constraints let you define custom validation rules that are enforced whenever data is inserted or updated. While typically used for value ranges and patterns, you can combine them with type-aware functions to create sophisticated type checks.

For example, you can validate that a string column contains only numeric characters:

```
CREATE TABLE orders (  
    order_id INT,  
    order_total STRING  
);  
  
ALTER TABLE orders  
ADD CONSTRAINT valid_order_total CHECK (order_total REGEXP '^[0-9]+(\\.[0-9]+)?$')
```

This constraint ensures the `order_total` column contains values that look like valid numbers, catching malformed data before it enters the table.

For date validation, you can use the `try_cast` function within a constraint:

```
ALTER TABLE events  
ADD CONSTRAINT valid_event_date CHECK (try_cast(event_date_str AS DATE) IS NOT NULL)
```

This approach rejects records where `event_date_str` doesn't contain a valid date format.

Important

Before adding a constraint to an existing table, Azure Databricks verifies that all existing rows satisfy the constraint. Plan for this validation step, especially on large tables.

Handle type mismatches in pipelines

When processing data from external sources, type mismatches are common. Implement a pattern that separates valid from invalid records, allowing you to process good data while quarantining problematic records for review:

```
-- Insert valid records into the target table  
INSERT INTO silver_transactions  
SELECT  
    transaction_id,  
    cast(amount AS DECIMAL(10,2)) AS amount,  
    cast(transaction_date AS DATE) AS transaction_date  
FROM bronze_transactions  
WHERE try_cast(amount AS DECIMAL(10,2)) IS NOT NULL  
    AND try_cast(transaction_date AS DATE) IS NOT NULL;
```

```
-- Capture invalid records for investigation
INSERT INTO quarantine_transactions
SELECT
    transaction_id,
    amount AS raw_amount,
    transaction_date AS raw_date,
    current_timestamp() AS quarantined_at,
    'Type validation failed' AS reason
FROM bronze_transactions
WHERE try_cast(amount AS DECIMAL(10,2)) IS NULL
    OR try_cast(transaction_date AS DATE) IS NULL;
```

This pattern ensures your pipeline continues processing valid data while preserving invalid records for later analysis.

Use pipeline expectations for type checking

Lakeflow Spark Declarative Pipelines provides **expectations** that allow you to define data quality rules directly in your pipeline definitions. You can use expectations to check that values can be cast to expected types:

```
from pyspark import pipelines as dp

@dp.table
@dp.expect_or_drop("valid_amount", "try_cast(amount AS DECIMAL(10,2)) IS NOT NULL")
@dp.expect_or_drop("valid_date", "try_cast(event_date AS DATE) IS NOT NULL")
def validated_transactions():
    return spark.readStream.table("raw_transactions")
```

With `expect_or_drop`, records that fail the type check are dropped before reaching the target table. Use `expect` to log violations without dropping records, or `expect_or_fail` to stop the pipeline when violations occur.

For SQL-based pipelines:

```
CREATE OR REFRESH STREAMING TABLE validated_transactions (
    CONSTRAINT valid_amount EXPECT (try_cast(amount AS DECIMAL(10,2)) IS NOT NULL)
    CONSTRAINT valid_date EXPECT (try_cast(event_date AS DATE) IS NOT NULL) ON VIOLATION
)
AS SELECT * FROM STREAM(raw_transactions);
```

Pipeline expectations integrate data type validation directly into your ETL logic, providing visibility into data quality metrics through the pipeline UI.

Implementing data type checks at multiple levels—schema enforcement, explicit casting, constraints, and pipeline expectations—creates defense in depth. Each layer catches issues that might slip past others, resulting in higher data quality in your Unity Catalog tables.

4. Detect and manage schema drift

<https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/4-detect-manage-schema-drift>

Detect and manage schema drift

Completed

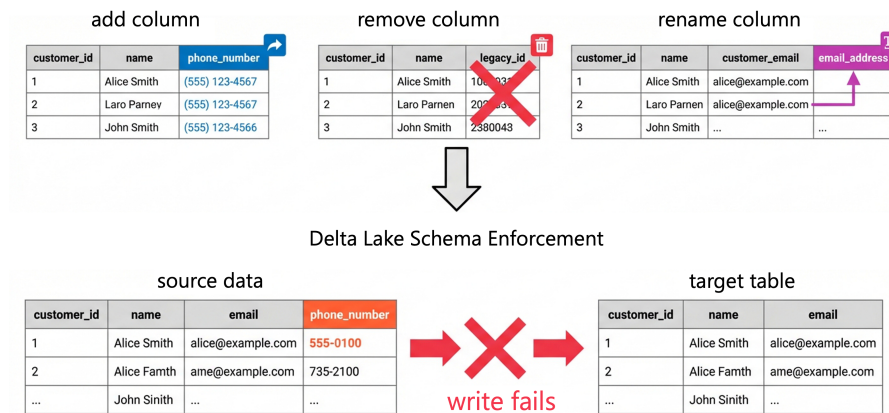
Data pipelines often receive data from sources that evolve over time. New columns appear, others disappear, and the structure of incoming data changes as business requirements shift. Without proper controls, these changes can silently corrupt your data or break your pipelines entirely.

In this unit, you learn how to detect and manage schema drift—the structural changes that occur when source systems add, remove, or rename columns over time.

Recognize schema drift challenges

While data type validation ensures values match expected types (as covered in the previous unit), schema drift addresses a different challenge: the structure of your data changes over time. A source system adds a new `phone_number` column, removes a deprecated `legacy_id` field, or renames `customer_email` to `email_address`. These structural changes happen independently of type validation.

Delta Lake's schema enforcement blocks structural mismatches by default. When incoming data contains columns not present in the target table, or when required columns are missing, the write operation fails. This fail-fast behavior protects your tables from unexpected structural changes, but you need strategies to handle legitimate schema evolution.



Consider a streaming pipeline that processes customer data:

```
CREATE TABLE customers (
  customer_id INT,
  name STRING,
  email STRING
);
```

When your source system adds a `phone_number` column and starts sending it in the data feed, writes fail because the target table doesn't include this column. Your pipeline stops until you decide how to handle the new field—either by rejecting it, adding it to the table schema, or preserving it for later analysis.

Detect and respond to schema drift

Schema drift occurs when the structure of incoming data changes over time. Your data pipelines must detect these changes and respond appropriately—either by adapting to the new schema or by alerting operators to investigate.

Azure Databricks provides several approaches for handling schema drift depending on your requirements.

Fail on schema mismatch

The strictest approach stops processing when schema drift is detected. This strategy works well when data structure changes require manual review before pipeline updates.

With Auto Loader, you can configure this behavior using the `failOnNewColumns` mode:

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")
  .option("cloudFiles.schemaLocation", "/path/to/schema")
  .option("cloudFiles.schemaEvolutionMode", "failOnNewColumns"))
```

```
.load("/path/to/source")
.writeStream
.option("checkpointLocation", "/path/to/checkpoint")
.toTable("target_table")
)
```

When Auto Loader encounters a new column not defined in the schema, the stream stops with an `UnknownFieldException`. You must update the schema or remove the offending data before the pipeline can resume.

Automatically adapt to new columns

For scenarios where adding new columns is acceptable, you can enable schema evolution. This approach automatically adds new columns to the target table without manual intervention.

Enable schema evolution for write operations:

```
(df.write
  .format("delta")
  .mode("append")
  .option("mergeSchema", "true")
  .saveAsTable("target_table")
)
```

For `MERGE` operations, use the schema evolution syntax:

```
MERGE WITH SCHEMA EVOLUTION INTO target_table t
USING source_data s
ON t.customer_id = s.customer_id
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
```

New columns from the source are automatically added to the target table. However, you should carefully consider when to use schema evolution—automatic changes might introduce columns you didn't expect.

Rescue unexpected data

When you need to preserve data that doesn't match your schema without modifying the target structure, use the rescued data column. This approach captures mismatched data in a separate JSON column for later review.

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")
  .option("cloudFiles.schemaEvolutionMode", "rescue")
  .option("rescuedDataColumn", "_rescued_data")
  .load("/path/to/source")
  .writeStream
```

```
.option("checkpointLocation", "/path/to/checkpoint")  
.toTable("target_table")  
)
```

Data that doesn't match the schema—including new columns, type mismatches, and case differences—is stored in the `_rescued_data` column. Your pipeline continues processing without interruption, and you can analyze rescued data to understand schema changes.

Implement error handling strategies

Beyond automatic schema management, you need strategies for handling records that violate your data quality rules. Lakeflow Spark Declarative Pipelines provides expectations that let you define constraints and specify how to handle violations.

Drop invalid records

Use expectations to filter out records that don't meet your structure requirements:

```
import databricks.sdk.pipelines as dp  
  
@dp.table  
@dp.expect_or_drop("required_columns_present",  
    "customer_id IS NOT NULL AND email IS NOT NULL")  
def validated_customers():  
    return spark.readStream.table("raw_customers")
```

Records missing required columns are dropped before reaching the target table. The pipeline tracks metrics showing how many records were dropped.

Fail on critical violations

When data structure issues indicate a serious problem, configure expectations to halt processing:

```
CREATE OR REFRESH STREAMING TABLE validated_orders (  
    CONSTRAINT valid_structure  
    EXPECT (order_id IS NOT NULL AND product_id IS NOT NULL)  
    ON VIOLATION FAIL UPDATE  
) AS SELECT * FROM STREAM(raw_orders);
```

The pipeline stops immediately when it encounters records missing required columns, preventing invalid data from propagating downstream.

Quarantine problematic records

For a balanced approach, route invalid records to a separate quarantine table while allowing valid records to proceed:

```
import databricks.sdk.pipelines as dp

@dp.table
def valid_transactions():
    return (
        spark.readStream.table("raw_transactions")
        .filter("amount IS NOT NULL AND transaction_date IS NOT NULL")
    )

@dp.table
def quarantine_transactions():
    return (
        spark.readStream.table("raw_transactions")
        .filter("amount IS NULL OR transaction_date IS NULL")
    )
```

This pattern lets you investigate problematic records without blocking your main data flow.

Apply best practices for schema management

Managing schema effectively requires a combination of technical controls and operational practices.

Document your expected schemas explicitly. Store schema definitions in version control alongside your pipeline code. When source systems change, you have a clear reference for understanding the impact.

Implement alerting for schema drift. Configure your pipelines to send notifications when schema changes are detected, even if the pipeline handles them automatically. Early awareness helps you understand evolving data patterns.

Use consistent naming conventions. Standardize column names across your data estate to reduce case-sensitivity issues and make schema validation more predictable.

Test schema handling in development. Before deploying pipelines to production, verify that your schema enforcement and drift handling work as expected with test data that includes edge cases.

Consider the tradeoffs between strictness and flexibility. Strict schema enforcement catches problems early but requires more maintenance. Flexible approaches like schema evolution reduce operational overhead but might allow unexpected changes.

With these strategies in place, your data pipelines can maintain data quality while adapting to the inevitable changes in your source systems.

5. Manage data quality with pipeline expectations

Manage data quality with pipeline expectations

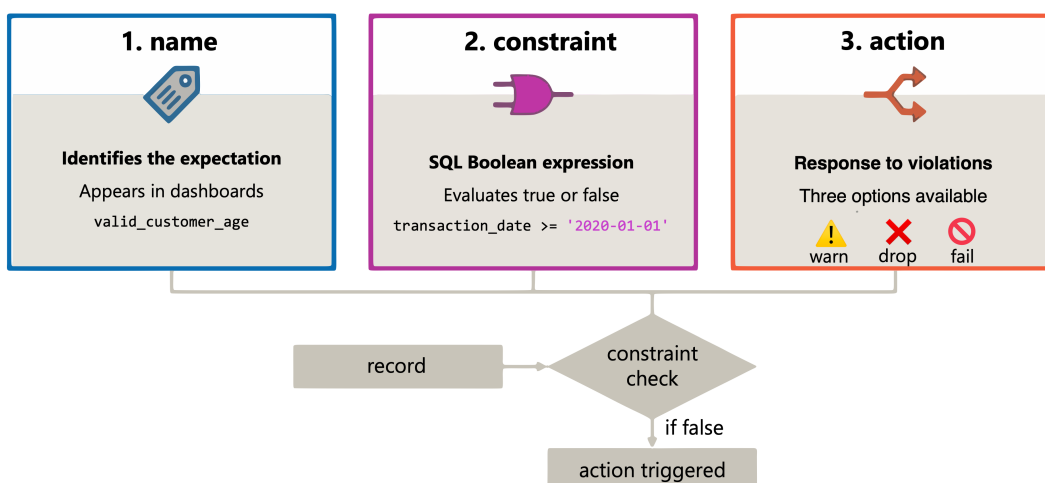
Completed

Data quality issues can disrupt downstream analytics and erode trust in your data. When invalid records slip into production tables, they might cause report failures, incorrect business decisions, or compliance violations. Pipeline expectations in Lakeflow Spark Declarative Pipelines give you a way to define quality rules that run automatically as data flows through your pipeline.

With expectations, you specify what valid data looks like using SQL constraints. The pipeline then checks every record against these rules and takes the action you configure—logging the issue, dropping the record, or failing the update entirely.

Define expectations with three components

Every expectation consists of three parts: a name, a constraint, and an action. Understanding these components helps you design effective data quality checks.



The **name** identifies the expectation and appears in monitoring dashboards. Choose names that clearly describe what you're validating. For example, `valid_customer_age` communicates the rule's purpose better than `check_1`.

The **constraint** is a SQL Boolean expression that evaluates to true or false for each record. When a record fails the constraint, the expectation triggers. You can use any valid SQL syntax except custom Python functions, external service calls, or subqueries.

Here's how you might check that transaction dates fall within an expected range:

```
@dp.expect("valid_transaction_date", "transaction_date >= '2020-01-01' AND transact
```

The same constraint in SQL:

```
CONSTRAINT valid_transaction_date EXPECT (transaction_date >= '2020-01-01' AND tra
```

The **action** determines what happens when a record violates the constraint. You have three options, each suited to different scenarios in your data pipeline.

Choose the right action for invalid records

Your choice of action depends on how critical the data quality issue is and how you want to handle violations.

Action	When to use	Behavior
Warn	Monitoring quality trends without blocking data	Invalid records flow through; metrics captured for review
Drop	Filtering out bad records automatically	Invalid records removed; valid records continue
Fail	Stopping the pipeline for critical violations	Update fails immediately; requires manual intervention

The **warn** action is the default behavior. It writes all records to the target table, including invalid ones, while logging metrics about failures. This approach works well when you're establishing baselines or when downstream systems can handle some data quality issues.

```
@dp.expect("non_null_email", "email IS NOT NULL")
def customer_contacts():
    return spark.readStream.table("raw.customers")
```

The **drop** action removes records that fail validation before writing to the target. Use this when you need clean data downstream and can afford to lose some records.

```
@dp.expect_or_drop("valid_price", "price >= 0 AND price <= 10000")
def validated_orders():
```

```
return spark.readStream.table("raw.orders")
```

The SQL equivalent uses `ON VIOLATION DROP ROW`:

```
CREATE OR REFRESH STREAMING TABLE validated_orders(  
    CONSTRAINT valid_price EXPECT (price >= 0 AND price <= 10000) ON VIOLATION DROP  
) AS SELECT * FROM STREAM(raw.orders);
```

The **fail** action stops execution immediately when any record violates the constraint. The pipeline atomically rolls back any partial updates. This option makes sense for critical data where invalid records indicate a serious upstream problem.

```
@dp.expect_or_fail("required_customer_id", "customer_id IS NOT NULL")  
def critical_transactions():  
    return spark.readStream.table("raw.transactions")
```

In SQL, use `ON VIOLATION FAIL UPDATE`:

```
CREATE OR REFRESH STREAMING TABLE critical_transactions(  
    CONSTRAINT required_customer_id EXPECT (customer_id IS NOT NULL) ON VIOLATION FAIL  
) AS SELECT * FROM STREAM(raw.transactions);
```

Note

When a pipeline has multiple parallel flows, a failure in one flow doesn't cause other flows to fail. Each flow operates independently.

Combine multiple expectations for comprehensive checks

Real-world data validation often requires multiple rules. You can stack expectations on a single dataset to check different aspects of data quality.

In Python, add multiple decorator calls:

```
@dp.expect("valid_amount", "amount > 0")  
@dp.expect("valid_currency", "currency IN ('USD', 'EUR', 'GBP')")  
@dp.expect("valid_timestamp", "created_at <= current_timestamp()")  
def validated_payments():  
    return spark.readStream.table("raw.payments")
```

Python also supports grouping expectations into reusable dictionaries. This approach helps you maintain consistency across multiple tables:

```
payment_rules = {  
    "valid_amount": "amount > 0",  
    "valid_currency": "currency IN ('USD', 'EUR', 'GBP')",
```

```

    "valid_timestamp": "created_at <= current_timestamp()"
  }

@dp.expect_all_or_drop(payment_rules)
def clean_payments():
    return spark.readStream.table("raw.payments")

```

The `expect_all`, `expect_all_or_drop`, and `expect_all_or_fail` decorators apply the same action to all expectations in the group.

In SQL, separate multiple constraints with commas:

```

CREATE OR REFRESH STREAMING TABLE validated_payments(
  CONSTRAINT valid_amount EXPECT (amount > 0),
  CONSTRAINT valid_currency EXPECT (currency IN ('USD', 'EUR', 'GBP')),
  CONSTRAINT valid_timestamp EXPECT (created_at <= current_timestamp())
) AS SELECT * FROM STREAM(raw.payments);

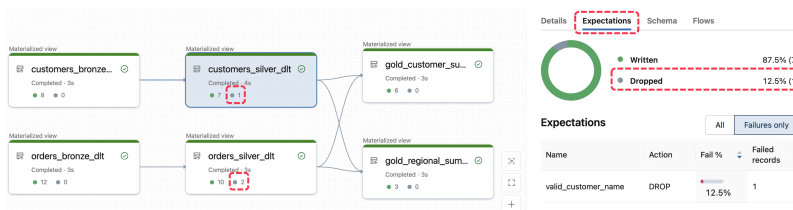
```

Monitor expectation results

After you define expectations, you can track their results through the pipeline UI. This visibility helps you understand data quality trends and identify recurring issues.

To view expectation metrics:

1. Open **Jobs & Pipelines** in the Azure Databricks workspace sidebar.
2. Select your pipeline by name.
3. Select a dataset that has expectations defined.
4. Open the **Data quality** tab in the right sidebar.



The metrics show you how many records passed or failed each expectation during pipeline runs. For `warn` and `drop` actions, you see counts of violations. For `fail` actions, the pipeline stops before metrics are recorded, but error messages include details about the violating record.

When a `fail` expectation triggers, the error message provides context to help you investigate:

```

[EXPECTATION_VIOLATION.VERBOSITY_ALL] Flow 'sensor-pipeline' failed to meet
the expectation. Violated expectations: 'temperature_in_valid_range'.
Input data: '{"id":"TEMP_001","temperature":-500,"timestamp_ms":"1710498600"}'.

```

You can also query expectation metrics programmatically through the Lakeflow Spark Declarative Pipelines event log. This approach enables you to build custom dashboards or trigger alerts based on

data quality thresholds.

Understanding when to use expectations sets you up to build pipelines that catch data quality issues before they affect downstream consumers. With the right combination of constraints and actions, you create a reliable data foundation for your analytics workloads.

6. Exercise - Implement and Manage Data Quality Constraints with Azure Databricks

<https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/6-exercise-implement-manage-data-quality-constraints-unity-catalog>

Exercise - Implement and Manage Data Quality Constraints with Azure Databricks

Completed

Now it's your chance to implement and manage data quality constraints with Azure Databricks. In this lab, you build a Lakeflow Spark Declarative Pipeline for ClearCover Insurance that enforces data quality constraints on raw claims data. You implement nullability and range checks using pipeline expectations, validate data types with `col().cast()`, and handle schema drift using Auto Loader's rescued data column. You then create and run the pipeline in the Databricks UI and monitor data quality metrics.

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](#)

7. Module assessment

Module assessment

Completed

8. Summary

<https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/8-summary>

Summary

Completed

Maintaining data quality requires a multi-layered approach that catches issues at every stage of your data pipeline. Throughout this module, you explored the tools Azure Databricks provides for implementing robust data quality constraints in Unity Catalog—from **schema enforcement** that validates data types automatically to **pipeline expectations** that monitor quality in real time.

You learned how Delta Lake's built-in schema enforcement acts as a first line of defense, rejecting writes when data types cannot be safely cast. For more nuanced type handling, **explicit casting** with `cast()` and `try_cast()` gives you control over how mismatches are resolved. **CHECK constraints** enforce business rules directly on your tables, ensuring invalid data never enters production.

Schema drift is inevitable when working with evolving data sources. You explored strategies for handling this reality—failing fast when changes require review, enabling **schema evolution** to adapt automatically, or using **rescued data columns** to preserve unexpected data for investigation. These options let you balance strictness with flexibility based on your pipeline's requirements.

Pipeline expectations in Lakeflow Spark Declarative Pipelines bring data quality validation directly into your ETL logic. You can **warn** about violations while still processing data, **drop** invalid records to ensure clean outputs, or **fail** pipelines when critical issues occur. The expectation metrics visible in the pipeline UI provide ongoing visibility into data quality trends.

Apply these techniques incrementally in your data engineering workflows. Start with schema enforcement and CHECK constraints for fundamental type and value validation. Add pipeline expectations for streaming workloads where real-time quality monitoring matters. Use quarantine

patterns to isolate problematic records without blocking your main data flows. Together, these practices create data pipelines that your organization can trust.