

Data Quality for Data Engineers

Frameworks, patterns, and tools for building reliable data pipelines.

Last reviewed 29 Sep 2026 · Lab-tested with Great Expectations 1.23.2 ·

Prerequisites: [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/)

Related: [dbt](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) · [Airflow](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) · [Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) · [Data Governance & Lineage](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/governance-lineage/) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/governance-lineage/) · [Testing and CI/CD](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/testing-cicd/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/testing-cicd/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

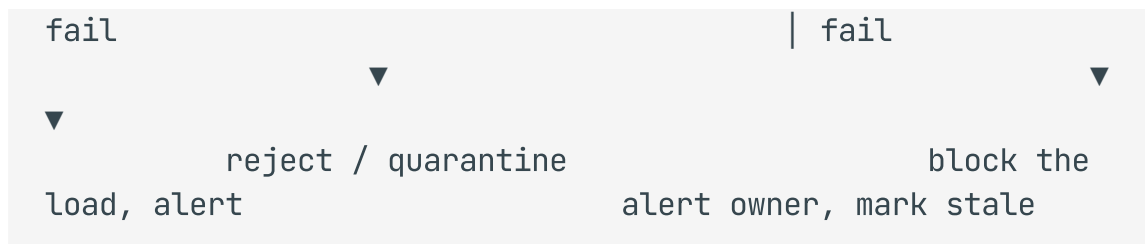
Practice: [Lab 01 — SQL Analytics](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/01-sql-analytics) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/01-sql-analytics) · [Lab 02 — dbt Transformations](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/02-dbt-transformations) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/02-dbt-transformations) · [Lab 05 — Airflow Orchestration](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/05-airflow-orchestration) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/05-airflow-orchestration)

Overview

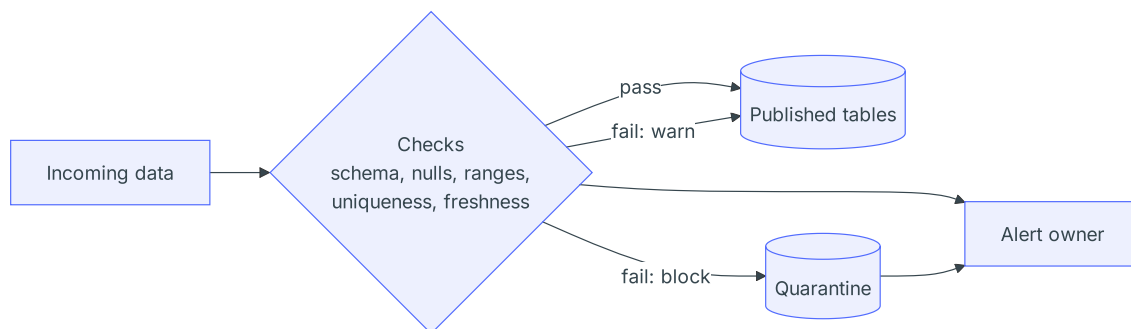
Challenge: Data pipelines rarely fail loudly. A source system renames a column, a partner delivers a partial file, or a join begins duplicating rows — and the pipeline still completes successfully. Without checks, the problem is discovered by business users, after decisions have already been made on incorrect data.

Solution: define what "correct" means for each dataset — no NULL identifiers, positive amounts, data delivered by an agreed time, row counts within the normal range — and verify it *automatically* on every run, blocking or flagging bad data before it is consumed.

```
Source → [contract/schema checks] → Bronze →  
[validity, uniqueness] → Silver → [business rules,  
reconciliation] → Gold → BI  
                    | fail
```



Relation to software testing: code tests verify logic at deployment time. Data tests must run on every load, because the data changes daily even when the code does not.



On this page

Basics - [Why Data Quality Matters](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#why-data-quality-matters) - [The Five Dimensions](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#the-five-dimensions) - [SQL-Based Checks](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#sql-based-checks) - [Python-Based Validation](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#python-based-validation)

Intermediate - [dbt Tests](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#dbt-tests) - [Great Expectations](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#great-expectations) - [Anomaly Detection Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#anomaly-detection-patterns) - [Data Contracts](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#data-contracts)

Advanced - [DQ in Streaming Pipelines](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#dq-in-streaming-pipelines) - [Observability & Alerting](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#observability--alerting) - [SLA Monitoring](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#sla-monitoring) - [Building a DQ Framework](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#building-a-dq-framework)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#interview-questions)

mbekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#interview-questions) -
[Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/#further-reading)

Why Data Quality Matters

Bad data is worse than no data — it leads to wrong decisions made with confidence.

```
Common failure modes:
  Upstream schema change → pipeline silently loads wrong
  types
  NULL in required column → downstream joins produce wrong
  counts
  Duplicate events          → revenue overstated by 12%
  Stale data                → dashboard shows yesterday's
  numbers as today's
  Row count drop (50%)     → partial load treated as
  complete
```

The cost of catching DQ issues late:

```
At ingestion:  fix the source, re-run — 30 minutes
At transform:  re-run transformations — 2 hours
At serving:    correct BI reports, notify stakeholders — 1
day
After decision: retract analysis, rebuild trust — weeks
```

The Five Dimensions

Dimension	Question	Example failure
Completeness	Is all expected data present?	Orders table missing 3 days of data
Accuracy	Does it reflect reality?	Negative order amounts after a sign error
Consistency	Does it agree across systems?	Revenue in warehouse $\neq$ revenue in source DB

Dimension	Question	Example failure
Timeliness	Is it fresh enough?	Daily report using yesterday's data at 9am
Uniqueness	Are there duplicates?	Customer counted twice after a dedup bug

SQL-Based Checks

The simplest DQ checks are SQL queries that return rows on failure. Zero rows = pass.

```
-- — Completeness —————
-- Required columns have no NULLs
SELECT COUNT(*) AS null_count
FROM orders
WHERE order_id IS NULL OR amount IS NULL;

-- Expected row count
SELECT COUNT(*) AS row_count FROM orders WHERE order_date =
'2024-03-15';
-- Alert if row_count < 1000 (or below historical average)

-- All expected sources present
SELECT DISTINCT source
FROM orders
WHERE order_date = CURRENT_DATE - 1
-- Alert if any expected source is missing from the result

-- — Accuracy —————
-- No negative amounts
SELECT order_id, amount
FROM orders
WHERE amount < 0;

-- Valid status values
SELECT DISTINCT status FROM orders
WHERE status NOT IN ('placed', 'shipped', 'delivered',
'cancelled', 'returned');

-- Age within realistic bounds
SELECT customer_id, age
```

```

FROM customers
WHERE age < 0 OR age > 130;

-- Prices within expected range
SELECT product_id, price
FROM products
WHERE price <= 0 OR price > 100000;

-- — Consistency —————
-- Row count matches source
SELECT
    (SELECT COUNT(*) FROM warehouse.orders WHERE order_date
= '2024-03-15') AS wh_count,
    (SELECT COUNT(*) FROM source_db.orders WHERE
DATE(created_at) = '2024-03-15') AS src_count;
-- Alert if ABS(wh_count - src_count) > 0

-- Revenue reconciliation
SELECT
    ABS(wh.revenue - src.revenue) AS discrepancy,
    wh.revenue AS warehouse_revenue,
    src.revenue AS source_revenue
FROM (
    SELECT SUM(amount) AS revenue FROM warehouse.orders
WHERE order_date = '2024-03-15'
) wh,
(
    SELECT SUM(amount) AS revenue FROM source_db.orders
WHERE DATE(created_at) = '2024-03-15'
) src
WHERE ABS(wh.revenue - src.revenue) > 0.01; -- allow 1
cent rounding

-- — Uniqueness —————
-- No duplicate primary keys
SELECT order_id, COUNT(*) AS n
FROM orders
GROUP BY order_id
HAVING COUNT(*) > 1;

-- No duplicate events (by business key + timestamp window)
SELECT user_id, event_type, DATE_TRUNC('minute',

```

```

event_time) AS minute, COUNT(*)
FROM events
GROUP BY 1, 2, 3
HAVING COUNT(*) > 3; -- more than 3 identical events in a
minute = suspicious

-- — Timeliness —————
-- Data is fresh enough
SELECT MAX(updated_at) AS latest_record,
       DATEDIFF('hour', MAX(updated_at),
CURRENT_TIMESTAMP()) AS hours_old
FROM orders;
-- Alert if hours_old > 2

-- Referential integrity
SELECT o.order_id
FROM orders o
LEFT JOIN customers c ON o.customer_id = c.id
WHERE c.id IS NULL; -- orders with no matching customer

```

Python-Based Validation

```

import pandas as pd
from dataclasses import dataclass
from typing import Callable

@dataclass
class Check:
    name: str
    fn: Callable[[pd.DataFrame], bool]
    severity: str = "error" # "error" | "warning"

def run_checks(df: pd.DataFrame, checks: list[Check]) -> dict:
    results = []
    for check in checks:
        passed = check.fn(df)
        results.append({
            "check": check.name,
            "passed": passed,

```

```

        "severity": check.severity,
    })
    if not passed and check.severity == "error":
        raise ValueError(f"DQ check FAILED:
{check.name}")
    return results

# Define checks
checks = [
    Check("no_null_order_id",
        lambda df: df["order_id"].notnull().all()),

    Check("no_negative_amount",
        lambda df: (df["amount"] >= 0).all()),

    Check("valid_status",
        lambda df:
df["status"].isin(["placed", "shipped", "delivered", "cancelled", "returned"]).all()),

    Check("no_duplicate_order_ids",
        lambda df: df["order_id"].nunique() == len(df)),

    Check("row_count_above_minimum",
        lambda df: len(df) >= 1000,
        severity="warning"),

    Check("amount_in_range",
        lambda df: df["amount"].between(0,
100_000).all()),
]

df = pd.read_parquet("s3://bucket/orders/2024-03-15/")
results = run_checks(df, checks)
print(f"Passed: {sum(r['passed'] for r in
results)}/{len(results)}")

```

dbt Tests

See also [dbt Reference](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) for full dbt test coverage.

```
# Four built-in generic tests
models:
  - name: fct_orders
    columns:
      - name: order_id
        data_tests:
          - unique
          - not_null
      - name: status
        data_tests:
          - accepted_values:
              values: ['placed', 'shipped', 'delivered',
'cancelled']
      - name: customer_id
        data_tests:
          - relationships:
              to: ref('dim_customer')
              field: customer_id
```

Custom singular test

```
-- tests/assert_revenue_matches_source.sql
-- Fails if revenue discrepancy > $1
WITH warehouse AS (
  SELECT SUM(amount) AS revenue
  FROM {{ ref('fct_orders') }}
  WHERE order_date = CURRENT_DATE - 1
),
source AS (
  SELECT SUM(amount) AS revenue
  FROM {{ source('raw', 'orders') }}
  WHERE DATE(created_at) = CURRENT_DATE - 1
)
SELECT
  warehouse.revenue AS wh_revenue,
  source.revenue AS src_revenue,
```

```
ABS(warehouse.revenue - source.revenue) AS discrepancy
FROM warehouse, source
WHERE ABS(warehouse.revenue - source.revenue) > 1.0
```

dbt-expectations (extended tests)

```
columns:
  - name: amount
    data_tests:
      -
        dbt_expectations.expect_column_values_to_be_between:
          min_value: 0
          max_value: 100000
        - dbt_expectations.expect_column_mean_to_be_between:
          min_value: 50
          max_value: 500

  - name: created_at
    data_tests:
      -
        dbt_expectations.expect_column_values_to_be_of_type:
          column_type: timestamp_ntz

models:
  - name: fct_orders
    data_tests:
      -
        dbt_expectations.expect_table_row_count_to_be_between:
          min_value: 1000
      -
        dbt_expectations.expect_table_columns_to_match_ordered_list:
          column_list: [order_id, customer_id, amount,
            status, created_at]
```

Great Expectations

Great Expectations is a Python library for defining, running, and documenting data quality checks.

Core concepts

Concept	Definition
Expectation	A verifiable assertion about data (e.g. "order_id is never null")
Expectation Suite	A collection of expectations for one dataset
Checkpoint	Connects a data source + expectation suite + action (alert, save results)
Data Docs	Auto-generated HTML documentation of all suites and results
Validation Result	The output of running an expectation suite — pass/fail + stats

Setup and basic usage (GX 1.x)

```
import great_expectations as gx
import pandas as pd

# Context holds data sources, suites, validation
# definitions, and checkpoints.
# get_context() is ephemeral (in-memory) unless a GX
# project directory exists;
# use gx.get_context(mode="file") to persist config
# alongside your code.
context = gx.get_context()

# Data source → asset → batch definition (here: validate a
# whole DataFrame)
data_source =
context.data_sources.add_pandas(name="orders_source")
data_asset =
data_source.add_dataframe_asset(name="orders")
batch_def =
data_asset.add_batch_definition_whole_dataframe("daily_orders")

# Expectation suite — expectations are classes in GX 1.x
suite =
context.suites.add(gx.ExpectationSuite(name="orders_suite")
)
suite.add_expectation(gx.expectations.ExpectColumnToExist(c
```

```

column="order_id"))
suite.add_expectation(gx.expectations.ExpectColumnValuesToNotBeNull(column="order_id"))
suite.add_expectation(gx.expectations.ExpectColumnValuesToBeUnique(column="order_id"))
suite.add_expectation(gx.expectations.ExpectColumnValuesToNotBeNull(column="amount"))
suite.add_expectation(gx.expectations.ExpectColumnValuesToBeBetween(
    column="amount", min_value=0, max_value=100_000))
suite.add_expectation(gx.expectations.ExpectColumnValuesToBeInSet(
    column="status", value_set=["placed", "shipped", "delivered", "cancelled"]))
suite.add_expectation(gx.expectations.ExpectTableRowCountToBeBetween(
    min_value=1_000, max_value=10_000_000))
suite.add_expectation(gx.expectations.ExpectColumnPairValuesAToBeGreaterThanB(
    column_A="updated_at", column_B="created_at", or_equal=True))

# Validation definition = which data + which suite
validation = context.validation_definitions.add(
    gx.ValidationDefinition(name="orders_validation",
        data=batch_def, suite=suite)
)

# Checkpoint = validations + actions (update Data Docs, Slack, email, ...)
checkpoint = context.checkpoints.add(
    gx.Checkpoint(
        name="orders_checkpoint",
        validation_definitions=[validation],
        actions=[
            gx.checkpoint.UpdateDataDocsAction(name="update_data_docs")
        ],
    )
)

# Run against today's data
df = pd.read_parquet("s3://my-bucket/orders/2024-03-15/")

```

```
result = checkpoint.run(batch_parameters={"dataframe": df})
print(result.success)          # True / False
```

Common expectations

Shown in snake_case for brevity. In GX 1.x each is a class —

expect_column_values_to_not_be_null("order_id") becomes

gx.expectations.ExpectColumnValuesToNotBeNull(column="order_id").

```
# Column existence
expect_column_to_exist("order_id")

# Null checks
expect_column_values_to_not_be_null("order_id")
expect_column_proportion_of_unique_values_to_be_between("status",
min_value=0.0, max_value=0.01)

# Value set
expect_column_values_to_be_in_set("status",
["placed", "shipped", "delivered"])
expect_column_values_to_not_be_in_set("country_code",
["XX", "ZZ"])

# Range
expect_column_values_to_be_between("amount", min_value=0,
max_value=100000)
expect_column_mean_to_be_between("amount", min_value=10,
max_value=1000)
expect_column_stdev_to_be_between("amount", min_value=0,
max_value=500)

# Type
expect_column_values_to_be_of_type("order_id", "str")
expect_column_values_to_match_regex("email", r"^[a-zA-Z0-9_+-.]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-+.]+$")

# Table-level
expect_table_row_count_to_be_between(min_value=1000)
expect_table_columns_to_match_set({"order_id",
"customer_id", "amount", "status"})

# Type list
```

```
expect_column_values_to_be_in_type_list("created_at",
["datetime64[ns]", "Timestamp"])

# Multi-column
expect_column_pair_values_a_to_be_greater_than_b("updated_at", "created_at")
```

Thresholds and severity

Two parameters turn an expectation from a strict assertion into a policy. `mostly` allows a share of rows to violate it, and `severity` (`critical`, `warning` or `info`, default `critical`) says how much a failure matters. Two expectations on one column give a warning well before the pipeline has to stop:

```
import great_expectations as gx
import pandas as pd

context = gx.get_context()
asset =
context.data_sources.add_pandas("s").add_dataframe_asset("orders")
batch_def =
asset.add_batch_definition_whole_dataframe("whole")
suite =
context.suites.add(gx.ExpectationSuite(name="orders_suite")
)
suite.add_expectation(gx.expectations.ExpectColumnValuesToNotBeNull(column="customer_id", mostly=0.95))
suite.add_expectation(gx.expectations.ExpectColumnValuesToNotBeNull(
    column="customer_id", mostly=0.995,
    severity="warning"))
validation = context.validation_definitions.add(
    gx.ValidationDefinition(name="orders_validation",
    data=batch_def, suite=suite))

df = pd.DataFrame({"customer_id": [1.0] * 98 + [None] * 2})
# 2% missing
result = validation.run(batch_parameters={"dataframe": df})

blocking = [r for r in result.results
             if not r.success and
```

```
r.expectation_config.severity == "critical"]
print("success:", result.success, "| blocking:",
len(blocking), "| failed:", sum(not r.success for r in
result.results))
```

`result.success` is `False` as soon as *any* expectation fails, including a warning. A gate that should block only on critical failures therefore reads the severity of each result itself, as `blocking` does above, instead of checking `result.success`. [Lab 08](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/08-data-quality) (<https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/08-data-quality>) builds this gate and shows it stopping bad, stale and schema-changed data.

Anomaly Detection Patterns

Statistical checks that catch unusual-but-valid-looking data.

```
-- Row count anomaly – alert if today's count is below 3-
sigma of the past 30 days
WITH daily_counts AS (
    SELECT order_date, COUNT(*) AS n
    FROM orders
    WHERE order_date >= CURRENT_DATE - 31
    GROUP BY order_date
),
stats AS (
    SELECT
        AVG(n) AS mean_n,
        STDDEV(n) AS std_n
    FROM daily_counts
    WHERE order_date < CURRENT_DATE -- exclude today
)
SELECT d.order_date, d.n,
    s.mean_n,
    s.mean_n - 3 * s.std_n AS lower_bound,
    CASE WHEN d.n < s.mean_n - 3 * s.std_n THEN 'ALERT'
ELSE 'OK' END AS status
FROM daily_counts d, stats s
WHERE d.order_date = CURRENT_DATE;

-- Revenue anomaly – compare today with the same weekday
over the past 4 weeks
WITH today AS (
```

```

        SELECT SUM(amount) AS revenue FROM orders WHERE
order_date = CURRENT_DATE
    ),
    same_weekday AS (
        -- one row per prior same-
weekday date
        SELECT order_date, SUM(amount) AS revenue
        FROM orders
        WHERE order_date IN (CURRENT_DATE - 7, CURRENT_DATE -
14,
                                CURRENT_DATE - 21, CURRENT_DATE -
28)
        GROUP BY order_date
    ),
    baseline AS (
        SELECT AVG(revenue) AS avg_revenue FROM same_weekday
    )
SELECT
    today.revenue,
    baseline.avg_revenue,
    (today.revenue - baseline.avg_revenue) /
NULLIF(baseline.avg_revenue, 0) * 100 AS pct_change
FROM today, baseline;

```

```

# Python: Z-score based anomaly detection
import pandas as pd
import numpy as np

def detect_anomalies(df: pd.DataFrame, col: str,
window_days: int = 30, z_threshold: float = 3.0) ->
pd.DataFrame:
    """Flag rows where the value is beyond z_threshold
standard deviations from the rolling mean."""
    rolling = df[col].rolling(window=window_days,
min_periods=7)
    df["rolling_mean"] = rolling.mean()
    df["rolling_std"] = rolling.std()
    df["z_score"] = (df[col] - df["rolling_mean"]) /
df["rolling_std"]
    df["is_anomaly"] = df["z_score"].abs() > z_threshold
    return df

```

Data Contracts

A data contract is a formal agreement between a data producer and consumer defining the schema, semantics, SLAs, and quality guarantees of a dataset.

```
# data_contract.yaml - example
name: orders
version: "1.2.0"
description: "Transactional orders from the e-commerce platform"
owner: "orders-data@example.com"
updated_at: "2024-03-15"

schema:
  fields:
    - name: order_id
      type: string
      nullable: false
      unique: true
      description: "UUID v4 order identifier"

    - name: customer_id
      type: integer
      nullable: false
      description: "FK to dim_customer"

    - name: amount
      type: decimal(12,2)
      nullable: false
      constraints:
        min: 0
        max: 100000

    - name: status
      type: string
      nullable: false
      constraints:
        enum: [placed, shipped, delivered, cancelled,
returned]

    - name: created_at
      type: timestamp
```

```
    nullable: false

quality:
    row_count_min_daily: 10000
    freshness_sla_hours: 2
    completeness_min_pct: 99.9

sla:
    availability: "99.9%"
    freshness: "data available by 06:00 UTC"
    support_contact: "data-support@example.com"
```

DQ in Streaming Pipelines

```
# PySpark Structured Streaming with inline DQ checks
from pyspark.sql import functions as F

def validate_streaming_batch(df, epoch_id):
    """Called for each micro-batch – apply DQ checks before
    writing."""

    # Null check
    null_count =
df.filter(F.col("order_id").isNull()).count()
    if null_count > 0:
        raise ValueError(f"Batch {epoch_id}: {null_count}
null order_ids")

    # Negative amount check
    bad_amounts = df.filter(F.col("amount") < 0)
    if bad_amounts.count() > 0:
        # Route bad records to quarantine, not main table

    bad_amounts.write.mode("append").parquet("s3://bucket/quara
ntine/orders/")
    df = df.filter(F.col("amount") >= 0)

    # Write clean records

df.write.format("delta").mode("append").save("s3://bucket/s
```

```
ilver/orders/")

stream = (
    spark.readStream.format("kafka")
        .option("subscribe", "order-events")
        .load()
        .select(F.from_json(F.col("value").cast("string"),
order_schema).alias("d"))
        .select("d.*")
)

stream.writeStream \
    .foreachBatch(validate_streaming_batch) \
    .option("checkpointLocation",
"s3://checkpoints/orders/") \
    .start()
```

DLT expectations (Databricks)

```
import dlt
from pyspark.sql import functions as F

@dlt.table
@dlt.expect("valid_amount", "amount >= 0")
@dlt.expect_or_drop("has_order_id", "order_id IS NOT
NULL")
@dlt.expect_or_fail("valid_status", "status IN
('placed','shipped','delivered','cancelled')")
def orders_silver():
    return dlt.read_stream("orders_bronze") \
        .withColumn("amount",
F.col("amount").cast("double"))
```

Observability & Alerting

Log DQ results to a table

```
from datetime import datetime
```

```
def log_dq_result(spark, pipeline_name, table_name,
check_name,
                    passed, row_count, details=None):
    record = [{
        "pipeline":    pipeline_name,
        "table":       table_name,
        "check":       check_name,
        "passed":      passed,
        "row_count":   row_count,
        "details":     details,
        "checked_at":  datetime.utcnow().isoformat(),
    }]
    spark.createDataFrame(record) \
        .write.format("delta").mode("append") \
        .saveAsTable("ops.dq_results")
```

Alert on failure

```
import requests

def slack_alert(check_name: str, table: str, details: str,
webhook_url: str):
    msg = {
        "text": f":red_circle: *DQ Check Failed*\n"
                f"*Check:* `{check_name}`\n"
                f"*Table:* `{table}`\n"
                f"*Details:* {details}"
    }
    requests.post(webhook_url, json=msg)

def run_check_with_alert(df, check_fn, check_name, table,
webhook_url):
    passed = check_fn(df)
    if not passed:
        slack_alert(check_name, table,
                    f"Check failed for {len(df)} rows",
                    webhook_url)
        raise ValueError(f"DQ check failed: {check_name}")
```

SLA Monitoring

```
-- Track pipeline freshness SLA
-- Alert if MAX(updated_at) is older than the SLA threshold

WITH slas AS (
    SELECT 'orders' AS table_name, 2 AS sla_hours UNION
    ALL
    SELECT 'customers' AS table_name, 24 AS sla_hours UNION
    ALL
    SELECT 'events' AS table_name, 1 AS sla_hours
),
freshness AS (
    SELECT 'orders' AS table_name, MAX(updated_at) AS
latest_ts FROM orders UNION ALL
    SELECT 'customers' AS table_name, MAX(updated_at)
FROM customers UNION ALL
    SELECT 'events' AS table_name, MAX(event_time)
FROM events
)
SELECT
    s.table_name,
    f.latest_ts,
    s.sla_hours,
    DATEDIFF('hour', f.latest_ts, CURRENT_TIMESTAMP()) AS
hours_stale,
    CASE
        WHEN DATEDIFF('hour', f.latest_ts,
CURRENT_TIMESTAMP()) > s.sla_hours
        THEN 'SLA BREACHED'
        ELSE 'OK'
    END AS status
FROM slas s
JOIN freshness f USING (table_name)
ORDER BY hours_stale DESC;
```

Building a DQ Framework

A minimal production DQ framework has four components:

1. Check definitions – what to validate (SQL or Python)
2. Check runner – execute checks, capture results
3. Result store – persist results for trending and alerting
4. Alerting – notify on failures, SLA breaches, anomalies

```
# Minimal framework skeleton
from dataclasses import dataclass
from typing import Callable
from datetime import datetime
import pandas as pd

@dataclass
class DQCheck:
    name: str
    fn: Callable[[pd.DataFrame], bool]
    severity: str = "error" # "error" blocks pipeline;
                           "warning" logs only

class DQRunner:
    def __init__(self, pipeline: str, table: str,
                 alert_fn=None):
        self.pipeline = pipeline
        self.table = table
        self.alert_fn = alert_fn
        self.results = []

    def run(self, df: pd.DataFrame, checks: list[DQCheck])
    -> bool:
        all_passed = True
        for check in checks:
            try:
                passed = check.fn(df)
            except Exception as e:
                passed = False

            self.results.append({
                "pipeline": self.pipeline,
                "table": self.table,
                "check": check.name,
```

```

        "passed":    passed,
        "severity":  check.severity,
        "checked_at":
datetime.utcnow().isoformat(),
    })

    if not passed:
        if check.severity == "error":
            all_passed = False
        if self.alert_fn:
            self.alert_fn(check.name, self.table,
check.severity)

    return all_passed

    def raise_on_failure(self):
        failed = [r for r in self.results if not
r["passed"] and r["severity"] == "error"]
        if failed:
            names = ", ".join(r["check"] for r in failed)
            raise ValueError(f"DQ checks failed: {names}")

```

Common Pitfalls

Pitfall	Symptom	Fix
Only testing at the end (Gold)	Bad data already spread through every model before a test fails	Test at each layer: schema at ingestion, keys and validity in Silver, business rules in Gold
Hundreds of noisy warning-level tests	Alert fatigue — real failures get ignored	Fewer, meaningful checks; separate <i>blocking</i> (error) from <i>informational</i> (warn); give each alert an owner
Static thresholds (<code>row_count > 1000</code>)	False alarms on weekends and holidays; misses a slow decline	Compare with a trailing baseline (same weekday, rolling mean $\pm k\sigma$)
Checking only that tests <i>passed</i>	Silent pipelines — no data arrived, so no rows failed	Freshness and volume checks: "data for yesterday exists and is in the normal range"

Pitfall	Symptom	Fix
Tests that scan the whole table every run	DQ jobs cost more than the pipeline	Test the new partition/increment; run full scans weekly
Failing the whole load for a handful of bad rows	Nothing loads; downstream teams get no data at all	Quarantine bad rows to a side table, load the rest, alert on the quarantine rate
No record of past results	Can't answer "when did this start?" or spot trends	Persist every check result (table, check, status, value, run time)
DQ owned only by the data team	The same upstream breakage keeps coming back	Data contracts with producers: schema, semantics, SLAs, change process
Reconciling on counts only	Row counts match but amounts are wrong	Reconcile sums/checksums of key measures against the source

Cheat Sheet

Which check catches what

Dimension	Check	dbt	Great Expectations
Completeness	Required columns not NULL	<code>not_null</code>	<code>ExpectColumnValuesToNotBeNull</code>
Uniqueness	Primary key unique	<code>unique / dbt_utils.unique_combination_of_columns</code>	<code>ExpectColumnValuesToBeUnique</code>
Validity	Allowed values / ranges	<code>accepted_values / dbt_utils.accepted_range</code>	<code>ExpectColumnValuesToBeInSet / ...ToBeBetween</code>
Integrity	Foreign keys resolve	<code>relationships</code>	<code>ExpectColumnValuesToBeInSet</code> (from a lookup)
Timeliness	Data is fresh	<code>dbt source freshness</code>	<code>ExpectColumnMaxToBeBetween</code> on the timestamp

Dimension	Check	dbt	Great Expectations
Volume	Row count in normal range	<code>dbt_expectations.expect_table_row_count_to_be_between</code>	<code>ExpectTableRowCountToBeBetween</code>
Consistency	Matches the source	Singular test comparing sums	Custom query expectation
Schema	Expected columns and types	<code>Model contracts (contract: {enforced: true})</code>	<code>ExpectTableColumnsToMatchSet</code>

SQL checks to keep handy

```
-- Duplicates on the grain
SELECT id, COUNT(*) FROM t GROUP BY id HAVING COUNT(*) > 1;

-- NULL rate per column
SELECT COUNT(*) - COUNT(col) AS nulls, 1.0 * (COUNT(*) - COUNT(col)) / COUNT(*) AS null_rate FROM t;

-- Orphaned foreign keys
SELECT f.* FROM fact f LEFT JOIN dim d ON f.dim_id = d.id
WHERE d.id IS NULL;

-- Freshness in hours
SELECT DATEDIFF('hour', MAX(loaded_at), CURRENT_TIMESTAMP()) FROM t;

-- Today's volume vs the trailing 28-day average
SELECT COUNT(*) / (SELECT COUNT(*) / 28.0 FROM t
                    WHERE dt BETWEEN CURRENT_DATE - 28 AND
CURRENT_DATE - 1)
FROM t WHERE dt = CURRENT_DATE;
```

Severity guide: block the pipeline for broken keys, schema breaks, and missing data · warn for distribution drift and soft thresholds · quarantine individual bad rows

Tool landscape: in-pipeline tests (dbt tests, Great Expectations, Soda, Pandera for DataFrames) · declarative pipeline expectations (Databricks, Snowflake data metric functions) · data observability platforms (Monte Carlo, Metaplane, Elementary for dbt) · contracts (Data Contract Specification, ODCS)

Interview Questions

Q: What are the dimensions of data quality? A: The usual five are completeness (is everything there, no missing rows or NULLs), accuracy/validity (values are correct and within allowed ranges), consistency (the same fact agrees across systems), timeliness (fresh enough for its use), and uniqueness (no duplicates at the declared grain). Some frameworks add integrity (relationships hold) and conformity (formats and types). The point is to turn "good data" into specific, testable checks.

Q: Where in a pipeline would you put data quality checks? A: At every boundary. At ingestion: schema and contract checks, and quarantine of malformed records. In Silver: primary-key uniqueness, NULLs in required columns, accepted values, referential integrity. In Gold: business rules and reconciliation against sources. Plus freshness and volume monitoring on the outputs. Earlier checks are cheaper to fix and stop bad data before it spreads.

Q: What is a data contract? A: A versioned, explicit agreement between the team that produces a dataset and the teams that consume it: schema and types, semantics of each field, quality guarantees, freshness SLA, ownership, and how breaking changes are communicated. Ideally it's enforced in code — CI on the producer side fails if a change breaks the contract — so a schema change becomes a negotiated release rather than a 3am incident.

Q: How would you detect a problem when no single row is invalid, but the data is still wrong? A: Monitor aggregates rather than rows: row counts, sums of key measures, NULL rates, and distinct counts compared with a baseline (same weekday last weeks, rolling mean $\pm 3\sigma$). Add reconciliation checks against the source system. Data observability tools automate this with learned thresholds, but a few well-chosen SQL checks cover most cases.

Q: A data quality check fails in production at 5am. What do you do? A: First, contain it: if it's blocking, make sure downstream consumers aren't reading partial or bad data, and mark the dataset stale if needed. Then diagnose from the stored check results and lineage — which upstream change, file, or run caused it? Communicate status to affected consumers early. Fix the root cause (or reload with correct data), then backfill idempotently and rerun checks. Afterwards, add a check or contract that would have caught it sooner.

Q: Great Expectations or dbt tests — how do you choose? A: If transformations live in dbt, start with dbt tests: they sit next to the models, run as part of `dbt build`, and cover most needs with packages like `dbt-utils` and `dbt-expectations`. Use Great Expectations (or Soda, or Pandera) when data needs validating outside the warehouse — in Python or Spark pipelines, on files before loading, or when you want its rich profiling and Data Docs. Many teams use both at different layers.

Further Reading

- [dbt data tests](https://docs.getdbt.com/docs/build/data-tests) (https://docs.getdbt.com/docs/build/data-tests) and [model contracts](https://docs.getdbt.com/docs/mesh/govern/model-contracts) (https://docs.getdbt.com/docs/mesh/govern/model-contracts)
- [Great Expectations documentation](https://docs.greatexpectations.io/docs/) (https://docs.greatexpectations.io/docs/) (GX 1.x)
- [Soda Core](https://docs.soda.io/) (https://docs.soda.io/) — YAML-based checks as an alternative to GX
- [Pandera](https://pandera.readthedocs.io/) (https://pandera.readthedocs.io/) — schema validation for pandas and Polars DataFrames
- [Elementary](https://docs.elementary-data.com/) (https://docs.elementary-data.com/) — open-source data observability for dbt
- [Data Contract Specification](https://datacontract.com/) (https://datacontract.com/) — an open standard, with a CLI for testing contracts
- *Data Quality Fundamentals* — Barr Moses, Lior Gavish & Molly Vorwerck (O'Reilly)

Previous: [BI Tools](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/bi-tools/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/bi-tools/) • **Next:** [Data Governance & Lineage](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/governance-lineage/) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/governance-lineage/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

🕒 September 29, 2026