Lecture 05-2: Inspecting and Modifying Data

DATA 503: Fundamentals of Data Engineering

Lucas P. Cordova, Ph.D.

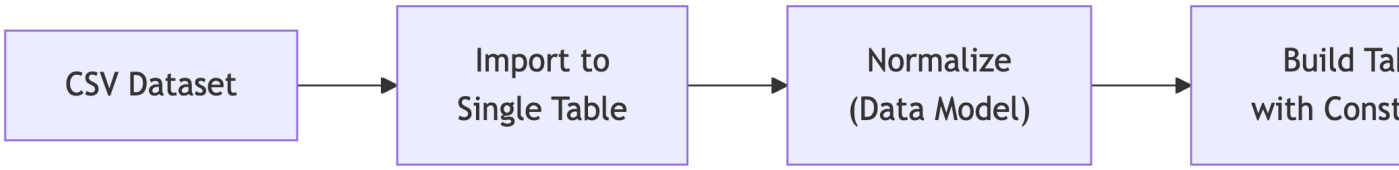
2026-02-09

This lecture covers the practical skills of inspecting, cleaning, and modifying data in PostgreSQL. After building normalized tables with constraints, we now learn to migrate data from a staging table into the final schema. Topics include auditing data quality, ALTER TABLE for schema changes, UPDATE for fixing data, DELETE for removing rows, backup strategies, and transactions for safe modifications. The music catalog dataset from the previous lecture serves as our running example.

Table of contents

1 Overview

1.1 Where We Are in the Pipeline

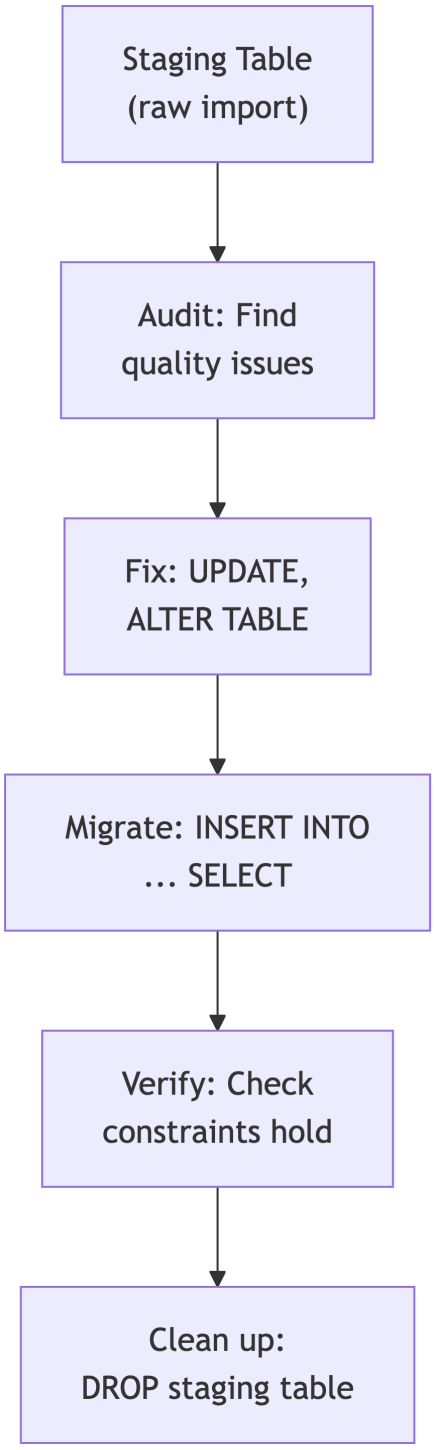


We built the tables. We added constraints. Now comes the part where we actually move the data from the staging table into the normalized schema without breaking anything. No pressure.

1.2 The Data Migration Challenge

Let's say you have a staging table (`music_catalog`) full of raw CSV data and a set of normalized tables (`artists`, `albums`, `tracks`) waiting to receive it. The problem is that raw data is rarely clean enough to insert directly.

The workflow:


Staging Table
(raw import)
Audit: Find
quality issues
Fix: UPDATE,
ALTER TABLE
Migrate: INSERT INTO
... SELECT
Verify: Check
constraints hold
Clean up:
DROP staging table

1.3 What You Will Learn Today

The DML (Data Manipulation Language) toolkit:

Statement	Purpose
`UPDATE`	Change existing values
`DELETE`	Remove rows
`ALTER TABLE`	Modify table structure
`INSERT INTO ... SELECT`	Copy data between tables
`START TRANSACTION`	Begin a safe modification block
`COMMIT`	Save changes permanently
`ROLLBACK`	Undo everything since the last transaction start

These are the verbs of data engineering. SELECT asks questions. These statements change answers.

2 Auditing Data Quality Process

2.1 The Inspection Mindset

Before modifying anything, inspect the data. Every experienced data engineer has a horror story about an UPDATE that ran without a WHERE clause. Do not become a horror story.

Questions to ask before any migration:

- How many rows do I have?
- Are there NULLs where there should not be?
- Are values consistent? (Same entity, different spellings?)
- Are there duplicates?
- Do the data types match the target schema?

2.2 Recall: The Music Catalog Staging Table

Last time, we imported a CSV of album data from a defunct record distributor and built our normalized target tables (`artists`, `albums`, `tracks`). The staging table looks like this:

```
1 CREATE TABLE music_catalog (
2     catalog_id varchar(20) CONSTRAINT catalog_key PRIMARY KEY,
3     artist_name varchar(200),
4     album_title varchar(200),
```

```
    release_year smallint,
    genre varchar(50),
    label varchar(100),
    duration_min numeric(5,1),
    decade varchar(10)
);
```

Six decades of music. Also six decades of data entry by people who apparently had strong opinions about capitalization.

2.3 Audit Step 1: How Many Rows?

Always start with the basics:

```
SELECT count(*) FROM music_catalog;
```

```
 count
-------
   609
```

This tells you the scale of what you are working with. A table with 50 rows and a table with 5 million rows require different strategies. One you can eyeball. The other, you cannot.

2.4 Audit Step 2: Find Duplicate Albums

```
SELECT artist_name,
       album_title,
       count(*) AS album_count
FROM music_catalog
GROUP BY artist_name, album_title
HAVING count(*) > 1
ORDER BY artist_name, album_title;
```

```
artist_name        album_title              album_count
-----------------  -----------------------  -----------
Radiohead          OK Computer                        2
The Beatles        Abbey Road                         2
```

`GROUP BY` with `HAVING count(*) > 1` is your duplicate detector. Same artist, same album, appearing twice. Could be a reissue. Could be a data entry mistake. Usually the latter.

2.5 Audit Step 3: Find NULL Values

```
SELECT genre,
       count(*) AS genre_count
FROM music_catalog
GROUP BY genre
ORDER BY genre;
```

```
genre          genre_count
-------------  -----------
Alternative             87
Country                 34
Electronic              45
Hip-Hop                 62
Pop                    128
R&B                     41
Rock                   203
                         9
```

Nine rows have no genre. That blank line at the bottom is NULL. NULL is not nothing. NULL is "I do not know," which in a database is significantly worse than nothing.

2.6 Audit Step 3b: Investigate the NULLs

```
SELECT catalog_id,
       artist_name,
       album_title,
       genre,
       release_year
FROM music_catalog
WHERE genre IS NULL;
```

`IS NULL` is the only way to check for NULL. Using `= NULL` will not work because NULL is not equal to anything, including itself. This is one of those things that makes perfect logical sense and no intuitive sense whatsoever.

2.7 Audit Step 4: Inconsistent Names

```
SELECT artist_name,
       count(*) AS name_count
FROM music_catalog
```

```
GROUP BY artist_name
ORDER BY artist_name ASC;
```

```
artist_name                 name_count
--------------------------  ----------
Led Zeppelin                         3
Led Zepplin                          1
Outkast                              2
OutKast                              1
the rolling stones                   1
The Rolling Stones                   4
Whitney Houston                      3
Whitney houston                      1
```

Eight entries for what should be four artists. Typos, inconsistent casing, and missing "The" prefixes. In a normalized database, each artist would be one row. In this staging table, each variation is a separate identity crisis.

2.8 Audit Step 5: Malformed Decade Values

```
SELECT decade,
       count(*) AS decade_count
FROM music_catalog
GROUP BY decade
ORDER BY decade;
```

```
decade      decade_count
----------  ------------
1970              42
1970s             58
1980s             89
1990s            104
2000s             97
2010s             78
20s               12
2020s             29
```

Three problems: "1970" should be "1970s", "20s" should be "2020s", and the two 1970s variants need merging. Whoever entered this data was consistent about 60% of the time, which is the worst kind of consistent.

2.9 Audit Step 5b: Check Release Years Against Decades

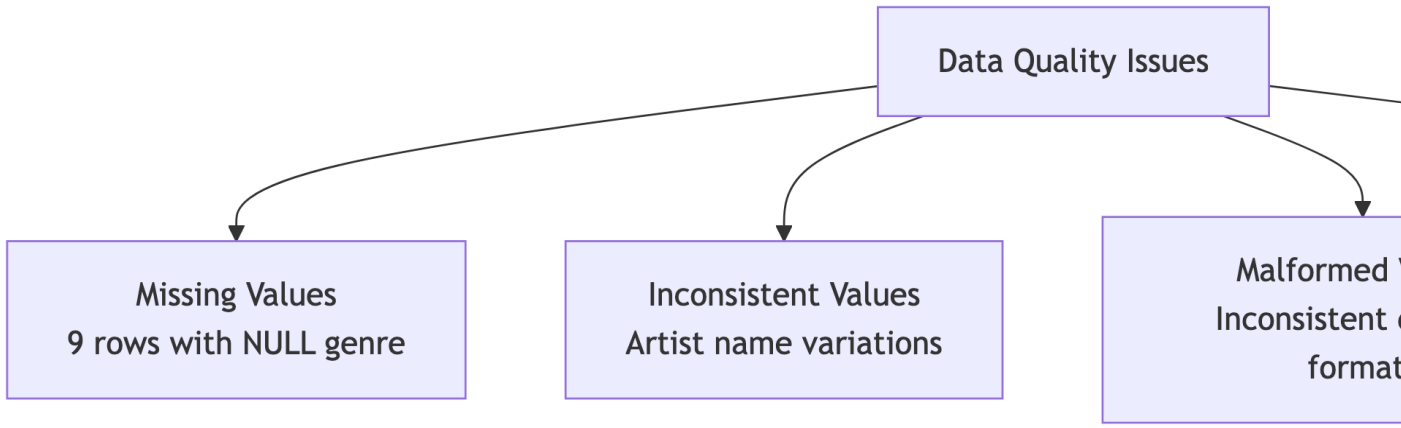
```
SELECT decade,
       min(release_year) AS earliest,
       max(release_year) AS latest
FROM music_catalog
GROUP BY decade
ORDER BY decade;
```

```
decade      earliest    latest
----------  --------    ------
1970              1970    1979
1970s             1970    1979
1980s             1980    1989
1990s             1990    1999
2000s             2000    2009
2010s             2010    2019
20s               2020    2024
2020s             2020    2025
```

At least the years match the decades. Small mercies.

2.10 Data Quality Audit Summary

Our audit revealed three categories of problems:



Now we fix them. But first, a word about safety.

3 The Golden Rule of Data Modification

3.1 Backup Before You Modify!

Always create a backup before running UPDATE or DELETE on production data.

This is not optional. This is not paranoia. This is professionalism. The difference between a junior and senior data engineer is not skill. It is the number of times they have been burned by a missing backup.

3.2 Creating a Backup Table

```
CREATE TABLE music_catalog_backup AS
SELECT * FROM music_catalog;
```

`CREATE TABLE ... AS SELECT` copies both structure and data. Verify the backup:

```
SELECT
    (SELECT count(*) FROM music_catalog) AS original,
    (SELECT count(*) FROM music_catalog_backup) AS backup;
```

```
original    backup
--------    ------
     609       609
```

Same count. You can proceed with slightly less anxiety.

3.3 The Safety Column Pattern

Before modifying a column, copy it first:

```
ALTER TABLE music_catalog ADD COLUMN artist_name_copy varchar(200);

UPDATE music_catalog
SET artist_name_copy = artist_name;
```

Now `artist_name_copy` preserves the original values. If your UPDATE goes sideways, you can restore from the copy without touching the backup table. Belt and suspenders.

4 Changing Structure

4.1 ALTER TABLE: Modifying Table Structure

`ALTER TABLE` changes the shape of a table without destroying its data:

Operation	Syntax
Add column	`ALTER TABLE t ADD COLUMN col type;`
Drop column	`ALTER TABLE t DROP COLUMN col;`
Change type	`ALTER TABLE t ALTER COLUMN col SET DATA TYPE type;`
Set NOT NULL	`ALTER TABLE t ALTER COLUMN col SET NOT NULL;`
Drop NOT NULL	`ALTER TABLE t ALTER COLUMN col DROP NOT NULL;`

4.2 Adding Columns for Data Cleaning

A common pattern: add a new "clean" column alongside the original dirty one.

```
-- Add a standardized artist name column
ALTER TABLE music_catalog
    ADD COLUMN artist_standard varchar(200);

-- Copy original values as starting point
UPDATE music_catalog
SET artist_standard = artist_name;
```

Now you can clean `artist_standard` without losing the original `artist_name` values. When you are satisfied the cleaning is correct, you can drop the original. Or keep both. Data engineers who keep both sleep better.

5 Updating and Fixing Data

5.1 The UPDATE Statement

`UPDATE` changes existing values in a table:

```
-- Update ALL rows (dangerous)
UPDATE table_name
SET column = value;

-- Update specific rows (safer)
```

```
UPDATE table_name
SET column = value
WHERE criteria;

-- Update multiple columns at once
UPDATE table_name
SET column_a = value_a,
    column_b = value_b
WHERE criteria;
```

> **Warning**
>
> An UPDATE without a WHERE clause modifies every row in the table. PostgreSQL will not ask "are you sure?" It will simply do it. Immediately. With enthusiasm.

5.2 Fixing Missing Genres

We found nine rows with NULL genres. After researching the albums, we can fill them in:

```
UPDATE music_catalog
SET genre = 'Rock'
WHERE catalog_id = 'CAT-4501';

UPDATE music_catalog
SET genre = 'Pop'
WHERE catalog_id = 'CAT-7823';

UPDATE music_catalog
SET genre = 'Hip-Hop'
WHERE catalog_id IN ('CAT-9102', 'CAT-9103', 'CAT-9104');
```

Each UPDATE returns `UPDATE 1` (or `UPDATE 3` for the IN clause), confirming the exact number of rows modified. If it says `UPDATE 0`, your WHERE clause matched nothing. If it says `UPDATE 609`, you forgot the WHERE clause. Both are worth investigating.

5.3 Restoring from a Safety Column

If your UPDATE was wrong, restore from the copy:

```
-- Option 1: Restore from the safety column
UPDATE music_catalog
SET artist_name = artist_name_copy;
```

```
4
5  -- Option 2: Restore from the backup table
6  UPDATE music_catalog original
7  SET artist_name = backup.artist_name
8  FROM music_catalog_backup backup
9  WHERE original.catalog_id = backup.catalog_id;
```

Option 2 uses `UPDATE ... FROM`, which joins two tables during the update. This is PostgreSQL-specific syntax and extremely useful for data migration work.

5.4 Standardizing Artist Names

Remember the inconsistent spellings? Fix them with pattern matching and exact matches:

```
1  -- Fix the typo
2  UPDATE music_catalog
3  SET artist_standard = 'Led Zeppelin'
4  WHERE artist_name LIKE 'Led Zep%';
5
6  -- Fix casing inconsistencies
7  UPDATE music_catalog
8  SET artist_standard = 'OutKast'
9  WHERE lower(artist_name) = 'outkast';
10
11 UPDATE music_catalog
12 SET artist_standard = 'The Rolling Stones'
13 WHERE lower(artist_name) = 'the rolling stones';
14
15 UPDATE music_catalog
16 SET artist_standard = 'Whitney Houston'
17 WHERE lower(artist_name) = 'whitney houston';
```

Verify:

```
1  SELECT artist_name, artist_standard
2  FROM music_catalog
3  WHERE artist_name != artist_standard
4  ORDER BY artist_standard;
```

```
artist_name            artist_standard
---------------------  --------------------
Led Zepplin            Led Zeppelin
the rolling stones     The Rolling Stones
```

```
OutKast                 OutKast
Outkast                 OutKast
Whitney houston         Whitney Houston
```

Five dirty values, four clean artist names. The original `artist_name` column is untouched for auditing.

5.5 Fixing Inconsistent Decade Values

The `||` operator concatenates strings in PostgreSQL. Combined with simple WHERE clauses, it handles our decade problems:

```sql
-- Fix "1970" -> "1970s"
UPDATE music_catalog
SET decade = '1970s'
WHERE decade = '1970';

-- Fix "20s" -> "2020s"
UPDATE music_catalog
SET decade = '2020s'
WHERE decade = '20s';
```

Verify all decades are now consistent:

```sql
SELECT decade, count(*) AS decade_count
FROM music_catalog
GROUP BY decade
ORDER BY decade;
```

```
decade      decade_count
----------  ------------
1970s             100
1980s              89
1990s             104
2000s              97
2010s              78
2020s              41
```

Six clean decades. No duplicates, no abbreviations.

5.6 Removing Duplicate Albums

For the duplicate Beatles and Radiohead entries, we need to keep one and remove the other. First, identify which to keep:

```
SELECT catalog_id, artist_name, album_title, release_year
FROM music_catalog
WHERE (artist_name, album_title) IN (
    SELECT artist_name, album_title
    FROM music_catalog
    GROUP BY artist_name, album_title
    HAVING count(*) > 1
)
ORDER BY artist_name, catalog_id;
```

Keep the one with the lower `catalog_id` (the first entry) and delete the duplicate:

```
DELETE FROM music_catalog
WHERE catalog_id IN ('CAT-1004', 'CAT-2087');
```

```
DELETE 2
```

Two rows removed. Verify the duplicates are gone:

```
SELECT artist_name, album_title, count(*)
FROM music_catalog
GROUP BY artist_name, album_title
HAVING count(*) > 1;
```

Empty result. Clean.

5.7 UPDATE with Subqueries

You can use a subquery to determine which rows to update:

```
ALTER TABLE music_catalog ADD COLUMN category varchar(50);

UPDATE music_catalog
SET category = 'Contemporary'
WHERE release_year >= 2000;

UPDATE music_catalog
SET category = 'Classic'
WHERE release_year < 2000;
```

5.8 UPDATE with FROM

A more readable alternative for cross-table updates:

```
UPDATE music_catalog m
SET category = g.category
FROM genre_tags g
WHERE m.genre = g.genre;
```

This joins `music_catalog` to `genre_tags` during the update and pulls the category value directly. Same result, cleaner syntax. This is PostgreSQL-specific and one of its nicest features.

5.9 UPDATE Patterns Summary

Pattern	Use When
`SET col = value WHERE ...`	Simple fixes to specific rows
`SET col = col2`	Copying between columns
`SET col = 'prefix' || col`	String manipulation
`UPDATE t1 SET ... FROM t2 WHERE ...`	Joining tables during update
`UPDATE ... WHERE EXISTS (SELECT ...)`	Conditional update based on another table
`WHERE lower(col) = 'value'`	Case-insensitive matching

6 Deleting Data

6.1 The DELETE Statement

`DELETE` removes rows from a table:

```
-- Delete ALL rows (very dangerous)
DELETE FROM table_name;

-- Delete specific rows (less dangerous)
DELETE FROM table_name WHERE expression;
```

> **Warning**
>
> `DELETE FROM table_name;` without a WHERE clause deletes every row. Unlike dropping the table, the empty table structure remains, staring at you like a reminder of what you

> have done.

6.2 Deleting Rows by Condition

```
DELETE FROM music_catalog
WHERE release_year < 1970;
```

```
DELETE 12
```

This removes all 12 albums released before the 1970s. PostgreSQL confirms the count, which you should always check against your expectation. If you expected 12 and got 12, good. If you expected 12 and got 412, less good.

6.3 DELETE vs DROP vs TRUNCATE

Statement	What It Does	Reversible?
`DELETE FROM t WHERE ...`	Removes matching rows	Yes (in a transaction)
`DELETE FROM t`	Removes all rows, keeps structure	Yes (in a transaction)
`TRUNCATE t`	Removes all rows, faster than DELETE	No
`DROP TABLE t`	Removes table entirely	No

7 Cleaning Up: DROP

7.1 Dropping Columns

After migration, remove temporary columns:

```
ALTER TABLE music_catalog DROP COLUMN artist_name_copy;
ALTER TABLE music_catalog DROP COLUMN category;
```

Clean tables are happy tables. Leftover columns named `_copy`, `_backup`, `_temp`, and `_old` are the database equivalent of packing boxes you never unpacked after moving.

7.2 Dropping Tables

Remove backup tables when you are confident the migration succeeded:

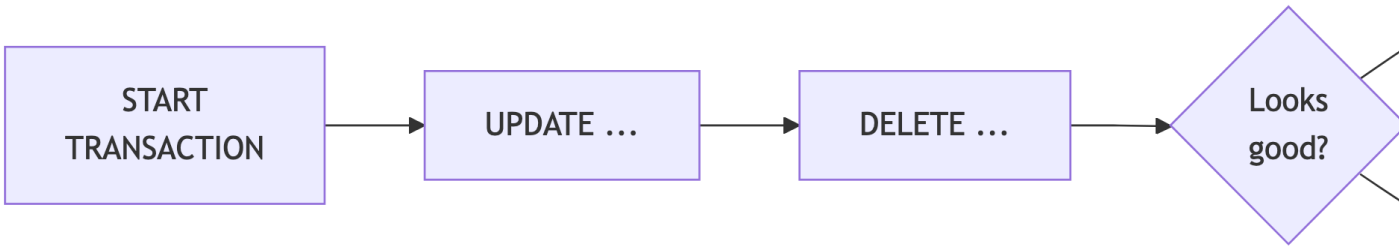
```
DROP TABLE music_catalog_backup;
```

`DROP TABLE` is permanent. There is no undo. Make sure you are done with the backup before dropping it. Then wait a day. Then drop it.

8 Transactions: Your Safety Net

8.1 What Is a Transaction?

A **transaction** groups multiple SQL statements into a single atomic unit. Either all of them succeed, or none of them do.



8.2 Transaction Syntax

```
START TRANSACTION;

UPDATE music_catalog
SET artist_standard = 'Feetwood Mac'
WHERE artist_name LIKE 'Fleetwood%';

-- Check your work
SELECT artist_name, artist_standard
FROM music_catalog
WHERE artist_name LIKE 'Fleetwood%';
```

```
11
12 -- Oops, typo! Undo everything.
13 ROLLBACK;
```

8.3 Transaction Example: Catching a Typo

```
1 START TRANSACTION;
2
3 UPDATE music_catalog
4 SET artist_standard = 'Feetwood Mac'   -- Note the missing 'l'
5 WHERE artist_name LIKE 'Fleetwood%';
```

Check the result:

```
artist_name       artist_standard
----------------  ----------------
Fleetwood Mac     Feetwood Mac     -- Typo!
Fleetwood Mac     Feetwood Mac     -- Typo!
Fleetwood Mac     Feetwood Mac     -- Typo!
```

The typo is visible. Roll it back:

```
1 ROLLBACK;
```

Query again:

```
artist_name       artist_standard
----------------  ----------------
Fleetwood Mac     Fleetwood Mac    -- Original restored
Fleetwood Mac     Fleetwood Mac
Fleetwood Mac     Fleetwood Mac
```

The ROLLBACK undid the UPDATE as if it never happened. Fleetwood Mac's name is safe. This is why transactions exist.

8.4 When to Use Transactions

Always use transactions when:

- Running multiple related UPDATE or DELETE statements
- Making changes you want to verify before committing
- Working on production data
- Performing data migrations

The workflow:

1. `START TRANSACTION;`
2. Run your statements
3. `SELECT` to verify the results
4. `COMMIT;` if correct, `ROLLBACK;` if not

> Tip
>
> In psql, `BEGIN` is an alias for `START TRANSACTION`. You will see both in the wild. They do the same thing.

8.5 Transaction Gotchas

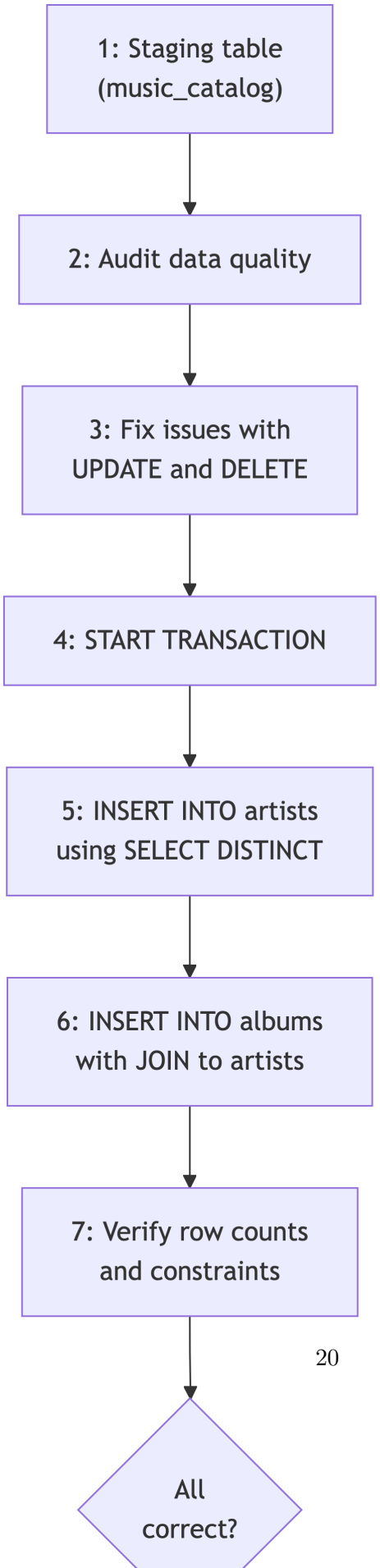
A few things to be aware of:

- DDL statements (`CREATE TABLE`, `DROP TABLE`) in PostgreSQL are transactional. This is unusual. Most other databases auto-commit DDL.
- If your session disconnects during an open transaction, PostgreSQL rolls it back automatically. Your data is safe, if slightly inconvenienced.
- Long-running open transactions can block other users. Do your work, then commit or rollback. Do not leave a transaction open while you go to lunch.

9 Data Migration: Putting It All Together

9.1 The Migration Workflow

With all these tools in hand, here is the complete workflow for migrating data from the staging table to our normalized music catalog:


1: Staging table (music_catalog)
2: Audit data quality
3: Fix issues with UPDATE and DELETE
4: START TRANSACTION
5: INSERT INTO artists using SELECT DISTINCT
6: INSERT INTO albums with JOIN to artists
7: Verify row counts and constraints
All correct?

9.2 INSERT INTO … SELECT

The key statement for migration copies data from one table to another:

```
-- Populate a normalized table from the staging table
INSERT INTO artists (artist_name)
SELECT DISTINCT artist_standard
FROM music_catalog
WHERE artist_standard IS NOT NULL
ORDER BY artist_standard;
```

`DISTINCT` ensures you do not insert duplicate rows. This is how you populate a lookup table from a denormalized staging table.

9.3 Migration with Foreign Keys

When migrating to tables with foreign key relationships, order matters:

```
START TRANSACTION;

-- 1. Populate parent table first (artists)
INSERT INTO artists (artist_name)
SELECT DISTINCT artist_standard
FROM music_catalog
WHERE artist_standard IS NOT NULL;

-- 2. Then populate child table with FK lookups (albums)
INSERT INTO albums (album_title, release_year, genre, label,
                    duration_min, artist_id)
SELECT DISTINCT m.album_title, m.release_year, m.genre, m.label,
       m.duration_min, a.artist_id
FROM music_catalog m
JOIN artists a ON m.artist_standard = a.artist_name;

-- 3. Verify
SELECT count(*) FROM artists;
SELECT count(*) FROM albums;

COMMIT;
```

Parent tables first, child tables second. Foreign key constraints will reject inserts in the wrong order.

9.4 Verifying the Migration

After committing, verify the data made it across cleanly:

```
-- Check counts
SELECT 'artists' AS tbl, count(*) FROM artists
UNION ALL
SELECT 'albums', count(*) FROM albums;

-- Spot-check with a JOIN
SELECT a.artist_name, al.album_title, al.release_year, al.genre
FROM albums al
JOIN artists a ON al.artist_id = a.artist_id
ORDER BY a.artist_name, al.release_year
LIMIT 10;
```

If the JOIN returns the data you expect, the migration worked. If it returns nothing, your foreign keys are not aligned. Check the ON clause.

9.5 Backup and Swap Pattern

A safer migration pattern creates the new table, verifies it, then swaps:

```
-- Create backup with extra metadata
CREATE TABLE music_catalog_archive AS
SELECT *,
       '2026-02-09'::date AS reviewed_date
FROM music_catalog;

-- Swap tables using RENAME
ALTER TABLE music_catalog
    RENAME TO music_catalog_temp;
ALTER TABLE music_catalog_archive
    RENAME TO music_catalog;
ALTER TABLE music_catalog_temp
    RENAME TO music_catalog_archive;
```

Three renames and the new table is live. The old one is still available as `_archive` if anything goes wrong.

10 Activity: Music Catalog Migration

10.1 The Scenario

You have the `music_catalog` staging table and the normalized `artists` and `albums` tables from the previous lecture. Your job: audit the staging data, fix the problems, and migrate the clean data into the normalized schema.

10.2 Part 1 : Audit the Data (5 min)

Write queries to find all the quality issues. You should check for:

- Duplicate albums (same artist + title appearing more than once)
- NULL values in columns that should not be empty
- Inconsistent artist name spellings and casing
- Malformed decade values

> Tip
>
> Use `GROUP BY ... HAVING count(*) > 1` for duplicates, `IS NULL` for missing values, and `GROUP BY` with `ORDER BY` to spot inconsistencies.

10.3 Part 2 : Back Up and Fix (10 min)

1. Create a backup table
2. Add a safety column for artist names
3. Fix each category of issues using UPDATE and DELETE:
    - Standardize artist name casing and fix typos
    - Fill in NULL genres (research the albums if needed)
    - Fix decade format inconsistencies (“1970” to “1970s”, “20s” to “2020s”)
    - Remove duplicate album entries

Remember: one category at a time, verify after each fix.

10.4 Part 3 : Migrate to Normalized Tables (10 min)

Write the migration wrapped in a transaction:

1. `INSERT INTO artists` using `SELECT DISTINCT` from the cleaned staging data
2. `INSERT INTO albums` with a JOIN back to `artists` for the foreign key
3. Verify row counts
4. COMMIT if correct, ROLLBACK if not

> **Tip**
>
> Parent tables first, child tables second. Your `albums` INSERT needs `artist_id` values, which means `artists` must be populated first.

10.5 One Possible Solution

10.5.1 Audit

```sql
-- Duplicates
SELECT artist_name, album_title, count(*)
FROM music_catalog
GROUP BY artist_name, album_title
HAVING count(*) > 1;

-- NULL genres
SELECT catalog_id, artist_name, album_title
FROM music_catalog
WHERE genre IS NULL;

-- Inconsistent artist names
SELECT artist_name, count(*)
FROM music_catalog
GROUP BY artist_name
ORDER BY artist_name;

-- Malformed decades
SELECT decade, count(*)
FROM music_catalog
GROUP BY decade
ORDER BY decade;
```

10.5.2 Backup and Fix

```sql
-- Backup
CREATE TABLE music_catalog_backup AS
SELECT * FROM music_catalog;

-- Safety column
ALTER TABLE music_catalog ADD COLUMN artist_standard varchar(200);
UPDATE music_catalog SET artist_standard = artist_name;

-- Fix artist names
UPDATE music_catalog
SET artist_standard = 'Led Zeppelin'
WHERE artist_name LIKE 'Led Zep%';

UPDATE music_catalog
SET artist_standard = 'OutKast'
WHERE lower(artist_name) = 'outkast';

UPDATE music_catalog
SET artist_standard = 'The Rolling Stones'
WHERE lower(artist_name) = 'the rolling stones';

UPDATE music_catalog
SET artist_standard = 'Whitney Houston'
WHERE lower(artist_name) = 'whitney houston';

-- Fix decades
UPDATE music_catalog SET decade = '1970s' WHERE decade = '1970';
UPDATE music_catalog SET decade = '2020s' WHERE decade = '20s';

-- Fix NULL genres (after research)
UPDATE music_catalog SET genre = 'Rock'
WHERE catalog_id = 'CAT-4501';
-- ... (remaining NULLs)

-- Remove duplicates
DELETE FROM music_catalog
WHERE catalog_id IN ('CAT-1004', 'CAT-2087');
```

10.5.3 Migration

```sql
START TRANSACTION;

-- 1. Artists
INSERT INTO artists (artist_name)
SELECT DISTINCT artist_standard
FROM music_catalog
WHERE artist_standard IS NOT NULL
ORDER BY artist_standard;

-- 2. Albums
INSERT INTO albums (album_title, release_year, genre, label,
                    duration_min, artist_id)
SELECT DISTINCT m.album_title, m.release_year, m.genre,
       m.label, m.duration_min, a.artist_id
FROM music_catalog m
JOIN artists a ON m.artist_standard = a.artist_name;

-- 3. Verify
SELECT 'artists' AS tbl, count(*) FROM artists
UNION ALL
SELECT 'albums', count(*) FROM albums;

COMMIT;
```

10.5.4 Verify

```sql
-- Full picture
SELECT a.artist_name, al.album_title,
       al.release_year, al.genre
FROM albums al
JOIN artists a ON al.artist_id = a.artist_id
ORDER BY a.artist_name, al.release_year;

-- Check constraints
SELECT * FROM albums WHERE release_year NOT BETWEEN 1900 AND 2100;
SELECT * FROM albums WHERE duration_min <= 0;
```

10.6 Part 4 : Discussion Questions

1. What would happen if you tried to insert albums before artists? What error would you see?
2. Why do we use `artist_standard` for the JOIN instead of `artist_name`?
3. If an artist later changes their name, how many rows need updating in the normalized schema vs the original flat staging table?
4. What would you add to make this migration idempotent (safe to run multiple times)?

11 Key Takeaways

11.1 The DML Toolkit

Tool	When to Use
`UPDATE ... SET ... WHERE`	Fix specific data quality issues
`UPDATE ... FROM`	Update using data from another table
`DELETE FROM ... WHERE`	Remove rows that do not belong
`ALTER TABLE ADD/DROP COLUMN`	Reshape tables for cleaning
`INSERT INTO ... SELECT`	Migrate data between tables
`START TRANSACTION / COMMIT / ROLLBACK`	Wrap everything in a safety net

11.2 The Data Engineer's Checklist

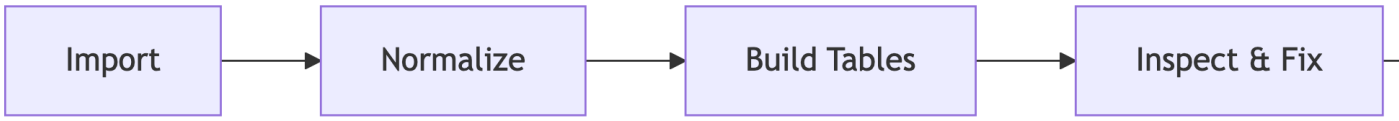
Before any data modification:

1. Audit the data thoroughly
2. Create a backup (table or safety columns)
3. Use transactions for multi-step changes
4. Verify after every modification
5. Keep the originals until you are certain

The goal is not speed. The goal is correctness. A fast migration that loses data is not a migration. It is a disaster with good performance metrics.

11.3 What Is Next

You now have the complete toolkit:



From raw CSV to a clean, normalized, constraint-enforced production database. The entire pipeline. Next, you will apply this pipeline end-to-end on your own with a new dataset.

11.4 References

11.5 Sources

1. DeBarros, A. (2022). *Practical SQL: A Beginner's Guide to Storytelling with Data* (2nd ed.). No Starch Press. Chapter 9: Inspecting and Modifying Data.
2. PostgreSQL Documentation. "UPDATE." https://www.postgresql.org/docs/current/sql-update.html
3. PostgreSQL Documentation. "DELETE." https://www.postgresql.org/docs/current/sql-delete.html
4. PostgreSQL Documentation. "Transactions." https://www.postgresql.org/docs/current/tutorial-transactions.html
5. PostgreSQL Documentation. "ALTER TABLE." https://www.postgresql.org/docs/current/sql-altertable.html