Lecture 08: Creating Tables and Constraints

DATA 351: Data Management with SQL

Lucas P. Cordova, Ph.D.

2026-02-16

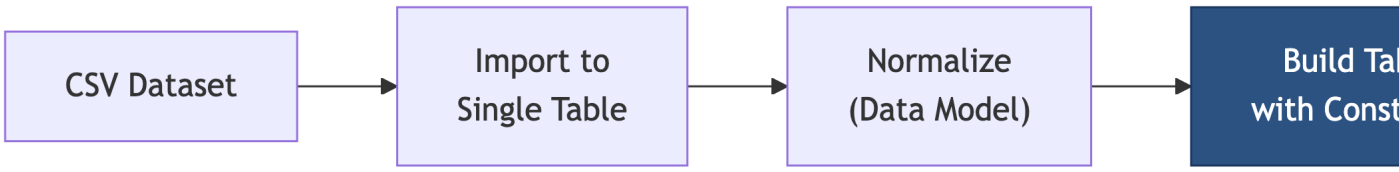
This lecture covers how to build database tables with proper constraints in PostgreSQL. After learning normalization, we now implement those designs with CREATE TABLE, primary keys, foreign keys, CHECK, UNIQUE, and NOT NULL constraints. We also cover ALTER TABLE for evolving schemas and indexes for query performance. A music catalog dataset serves as the running example.

Table of contents

1 The Big Picture

1.1 From Blueprint to Building

Last time we learned to normalize data. That was the blueprint. Today we pour the concrete.



1.2 Today's Agenda

We are learning DDL (Data Definition Language) to **implement** your normalized designs:

- Naming conventions (the surprisingly heated topic)
- Primary keys (natural vs surrogate)
- Foreign keys (relationships between tables)
- CHECK, UNIQUE, and NOT NULL constraints
- ALTER TABLE (because nobody gets it right the first time)
- Indexes (making queries go brrr)

2 Our Running Example: A Music Catalog

2.1 The Scenario

You work for a streaming service that just acquired a catalog of albums from a defunct record distributor. The data arrived as a single CSV spanning six decades of music, from Fleetwood Mac to Beyonce.

Your job: normalize it and build proper tables.

2.2 The Messy Data

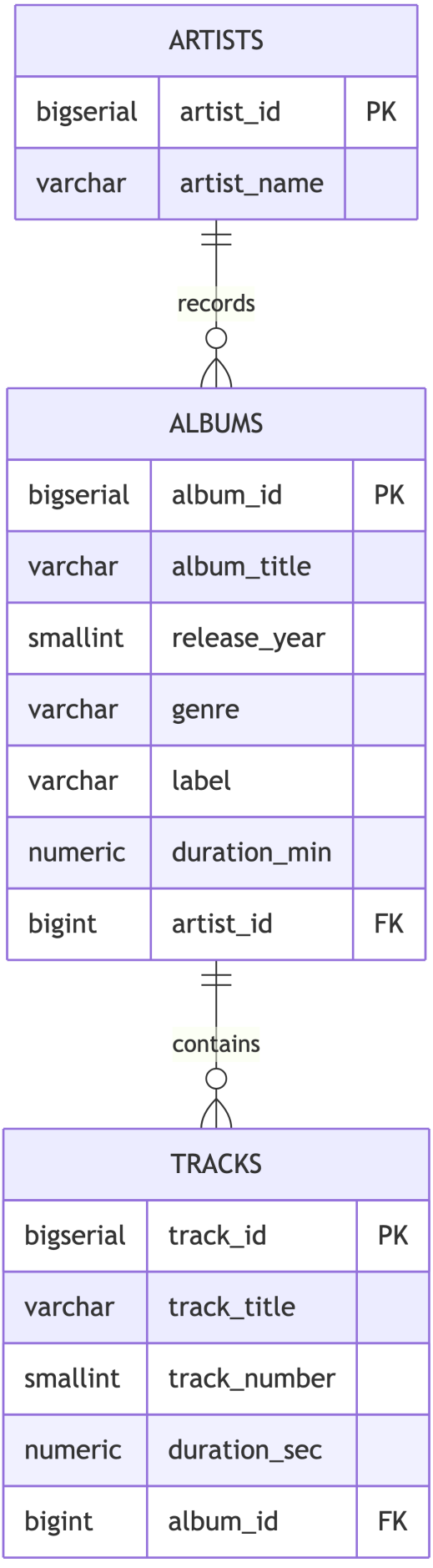
Here is a sample of what you received:

catalog_id	artist_name	album_title	release_year	genre	label	duration_min
CAT-1001	Fleetwood Mac	Rumours	1977	Rock	Warner Bros	39.4
CAT-1003	The Beatles	Abbey Road	1969	Rock	Apple	47.4
CAT-1004	The Beatles	Abbey Road	1969	Rock	Apple	47.4
CAT-1005	Led Zepplin	Led Zeppelin IV	1971	Rock	Atlantic	42.5
CAT-1006	Beyonce	Lemonade	2016	R&B	Columbia	45.7
CAT-1009	the rolling stones	Sticky Fingers	1971	Rock	Rolling Stones	46.3
CAT-1012	Outkast	Aquemini	1998	Hip-Hop	LaFace	72.6

Duplicates, typos, inconsistent casing… this data has seen things.

2.3 The Normalized Target

After normalization, we want three clean tables:


ARTISTS
bigserial
artist_id
PK
varchar
artist_name
records
ALBUMS
bigserial
album_id
PK
varchar
album_title
smallint
release_year
varchar
genre
varchar
label
numeric
duration_min
bigint
artist_id
FK
contains
TRACKS
bigserial
track_id
PK
varchar
track_title
smallint
track_number
numeric
duration_sec
bigint
album_id
FK

Today we learn how to **build** these tables. Next time we **fill** them.

3 Naming Conventions

3.1 Why Naming Matters

Good naming makes your database self-documenting. Bad naming leads to:

- Confusion when writing queries
- Bugs from misremembering column names
- Onboarding headaches for new team members
- Passive-aggressive comments in code reviews

Your future self is a team member. Be nice to them.

3.2 PostgreSQL Naming Rules

PostgreSQL identifiers (table and column names):

- Can contain letters, digits, and underscores
- Must begin with a letter or underscore
- Are **case-insensitive** by default (folded to lowercase)
- Maximum length of 63 characters

```
-- These all refer to the SAME table:
CREATE TABLE artists (...);
CREATE TABLE Artists (...);  -- Error: already exists!
CREATE TABLE ARTISTS (...);  -- Error: already exists!
```

PostgreSQL treats your SHOUTING the same as your whispering.

3.3 The Double-Quote Trap

If you use double quotes, the name becomes case-sensitive:

```
CREATE TABLE "Artists" (...);  -- Creates "Artists" (capital A)
SELECT * FROM artists;          -- Looks for "artists" (lowercase)
SELECT * FROM "Artists";        -- Finds "Artists" (capital A)
```

> **Warning**
>
> Avoid double-quoted identifiers. They create maintenance headaches because every query must match the exact casing with quotes. You will curse your past self at 2 AM.

3.4 Best Practices

Convention	Example	Avoid
Use snake_case	`release_year`	`releaseYear`, `ReleaseYear`
Be descriptive	`artist_name`	`art_nm`
Use plurals for tables	`artists`, `albums`	`artist`, `album`
Include units	`duration_min`	`duration`
Prefix dates	`report_2026_01_15`	`15_01_2026_report`

Tables are collections, so plural. Columns are attributes, so singular and descriptive.

3.5 Knowledge Check 1

Which table name follows PostgreSQL best practices?

A) `"AlbumData"`

B) `album_data`

C) `albumdata`

D) `Album-Data`

Think about: **Why** is your answer the best choice?

3.6 Answer: Knowledge Check 1

B) `album_data`

Snake_case, lowercase, descriptive, no quotes needed.

4 Primary Keys

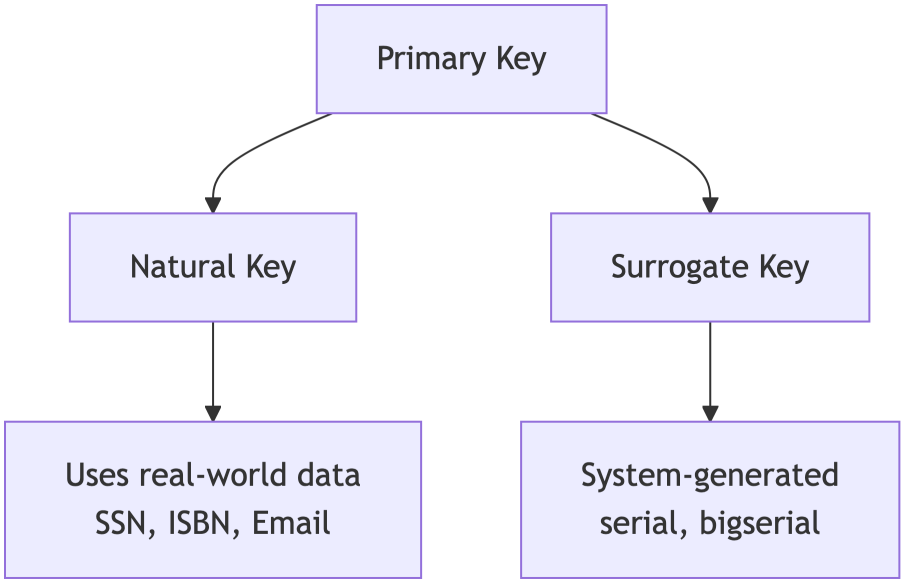
4.1 What Is a Primary Key?

A **primary key** uniquely identifies each row in a table. It guarantees:

- **Uniqueness:** No two rows share the same key value
- **Non-nullability:** The key cannot be NULL
- **Identity:** A reliable way to reference a specific row

Every well-designed table should have one.

4.2 Two Flavors of Primary Keys



4.3 Natural Keys

A **natural key** uses data that already exists and naturally identifies the entity:

```
CREATE TABLE natural_key_example (
    license_id varchar(10) CONSTRAINT license_key PRIMARY KEY,
    first_name varchar(50),
    last_name varchar(50)
);
```

Each person has exactly one license number and it is unique. In theory. In practice, natural keys have a habit of being less unique than you were promised.

4.4 Natural Keys: What Happens with Duplicates?

```
INSERT INTO natural_key_example (license_id, first_name, last_name)
VALUES ('T229901', 'Lynn', 'Malero');

INSERT INTO natural_key_example (license_id, first_name, last_name)
VALUES ('T229901', 'Sam', 'Tracy');
```

The second INSERT fails:

```
ERROR: duplicate key value violates unique constraint "license_key"
DETAIL: Key (license_id)=(T229901) already exists.
```

The database is polite about it, but firm.

4.5 Composite Natural Keys

Sometimes no single column is unique, but a **combination** is:

```
CREATE TABLE attendance (
    student_id varchar(10),
    school_day date,
    present boolean,
    CONSTRAINT student_key PRIMARY KEY (student_id, school_day)
);
```

A student can only have one attendance record per day. Neither column is unique alone, but together they form a unique identifier.

4.6 Surrogate Keys

A **surrogate key** is a system-generated value with no real-world meaning:

```
CREATE TABLE artists (
    artist_id bigserial,
    artist_name varchar(200) NOT NULL,
    CONSTRAINT artist_key PRIMARY KEY (artist_id)
);
```

PostgreSQL auto-generates incrementing integers:

Type	Range
`smallserial`	1 to 32,767
`serial`	1 to ~2.1 billion
`bigserial`	1 to ~9.2 quintillion

4.7 Surrogate Keys in Action

```
INSERT INTO artists (artist_name)
VALUES ('Fleetwood Mac'),
       ('The Beatles'),
       ('Beyonce');

SELECT * FROM artists;
```

```
 artist_id |  artist_name
-----------+---------------
         1 | Fleetwood Mac
         2 | The Beatles
         3 | Beyonce
```

We never specified `artist_id`. PostgreSQL handled it. One less thing to argue about.

4.8 Natural vs Surrogate: When to Use Which

Factor	Natural Key	Surrogate Key
Meaning	Has real-world meaning	Meaningless identifier
Stability	Can change (email, name)	Never changes
Size	Varies (could be long)	Fixed, compact
Performance	Depends on data type	Fast (integer)
Universality	Not always available	Always available

Tip

Rule of thumb: Use surrogate keys (`bigserial`) for most tables. If a natural key is truly stable (ISBN, SSN), consider it. When in doubt, surrogate wins.

4.9 Two Syntax Styles

4.9.1 Inline (Column Level)

```
CREATE TABLE artists (
    artist_id bigserial CONSTRAINT artist_key PRIMARY KEY,
    artist_name varchar(200) NOT NULL
);
```

Best for single-column keys.

4.9.2 Table Level

```
CREATE TABLE artists (
    artist_id bigserial,
    artist_name varchar(200) NOT NULL,
    CONSTRAINT artist_key PRIMARY KEY (artist_id)
);
```

Required for composite keys. Also works for single-column keys.

4.10 Knowledge Check 2

You are building a table to store student grades per course. Which primary key strategy works best?

A) Use the student's email as a natural key

B) Use a composite key of `(student_id, course_id)`

C) Use a `bigserial` surrogate key and a UNIQUE constraint on `(student_id, course_id)`

D) Use the student's name as a natural key

Think about: **Why** does your answer handle the real-world better than the others?

4.11 Answer: Knowledge Check 2

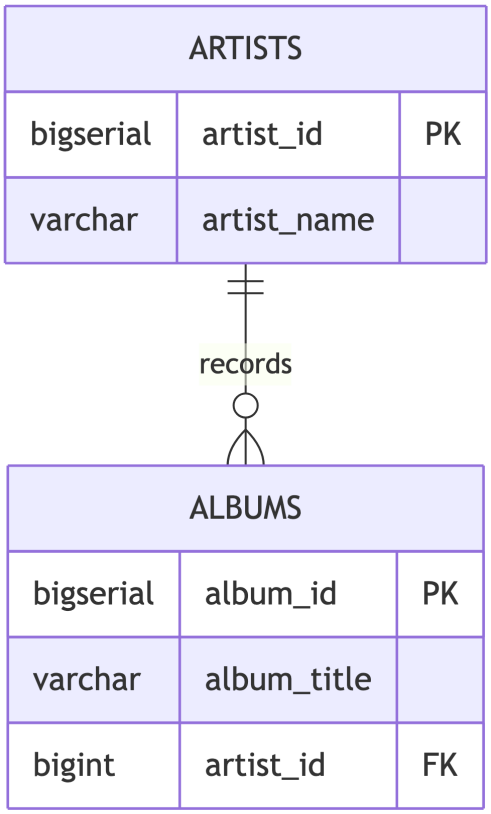
C) Use a `bigserial` surrogate key with a UNIQUE constraint on `(student_id, course_id)`

A surrogate key is stable, and the UNIQUE constraint still prevents duplicate enrollments.

5 Foreign Keys

5.1 Connecting Tables

A **foreign key** is a column in one table that references the primary key of another. It enforces **referential integrity**: you cannot reference a row that does not exist.



5.2 Creating Foreign Keys

```
CREATE TABLE artists (
    artist_id bigserial,
    artist_name varchar(200) NOT NULL,
    CONSTRAINT artist_key PRIMARY KEY (artist_id)
);

CREATE TABLE albums (
    album_id bigserial,
    album_title varchar(200) NOT NULL,
```

```
    release_year smallint,
    artist_id bigint REFERENCES artists (artist_id),
    CONSTRAINT album_key PRIMARY KEY (album_id)
);
```

The `REFERENCES` keyword creates the contract: "This value must exist in the other table." The database holds you to it.

5.3 Referential Integrity in Action

```
-- This works: artist_id 1 exists
INSERT INTO artists (artist_name) VALUES ('Fleetwood Mac');

INSERT INTO albums (album_title, release_year, artist_id)
VALUES ('Rumours', 1977, 1);
```

```
-- This FAILS: artist_id 999 does not exist
INSERT INTO albums (album_title, release_year, artist_id)
VALUES ('Phantom Album', 2025, 999);
```

```
ERROR: insert or update on table "albums" violates foreign key
constraint "albums_artist_id_fkey"
DETAIL: Key (artist_id)=(999) is not present in table "artists".
```

5.4 What Happens When You Delete a Parent?

By default, PostgreSQL **prevents** deleting a parent row if children reference it.

ON DELETE CASCADE auto-deletes children when the parent is removed:

```
CREATE TABLE tracks (
    track_id bigserial,
    track_title varchar(200) NOT NULL,
    album_id bigint REFERENCES albums (album_id)
        ON DELETE CASCADE,
    CONSTRAINT track_key PRIMARY KEY (track_id)
);
```

Delete an album and all its tracks vanish with it.

5.5 ON DELETE Options

Option	Behavior
`RESTRICT` (default)	Prevent deletion if children exist
`CASCADE`	Delete children automatically
`SET NULL`	Set foreign key to NULL in children
`SET DEFAULT`	Set foreign key to default value

> ⚠ Warning
>
> Use CASCADE carefully. Deleting one artist could cascade through albums and tracks, removing far more data than intended. CASCADE is the database equivalent of pulling a loose thread on a sweater.

5.6 Knowledge Check 3

You have an `orders` table with a foreign key to `customers`. A customer wants to delete their account, but they have existing orders. What does the DEFAULT behavior do?

A) Deletes the customer and all their orders

B) Deletes the customer and sets `customer_id` to NULL in orders

C) Prevents deleting the customer

D) Deletes the customer but keeps the orders unchanged

Think about: **Why** is this the safest default?

5.7 Answer: Knowledge Check 3

C) Prevents deleting the customer

The default is `RESTRICT`. It protects data by refusing to leave orphan rows.

6 CHECK Constraints

6.1 Validating Data at the Gate

A `CHECK` constraint ensures column values meet a condition. If the condition is false, the row is rejected. No exceptions. No bribes.

```
CREATE TABLE albums (
    album_id bigserial,
    album_title varchar(200) NOT NULL,
    release_year smallint,
    duration_min numeric(5,1),
    CONSTRAINT album_key PRIMARY KEY (album_id),
    CONSTRAINT check_year_range
        CHECK (release_year BETWEEN 1900 AND 2100),
    CONSTRAINT check_duration_positive
        CHECK (duration_min > 0)
);
```

6.2 CHECK: Practical Examples

```
-- Genre must be from a known list
CONSTRAINT check_genre
    CHECK (genre IN ('Rock', 'Pop', 'Hip-Hop', 'R&B',
           'Country', 'Electronic', 'Alternative', 'Jazz'))

-- Track number must be positive
CONSTRAINT check_track_positive
    CHECK (track_number > 0)

-- Duration must be within reason (no 2-hour tracks)
CONSTRAINT check_duration
    CHECK (duration_sec BETWEEN 1 AND 7200)
```

> **Tip**
>
> CHECK constraints catch bad data at the database level, regardless of which application inserts it. Applications come and go, but the database remembers.

7 UNIQUE Constraints

7.1 Beyond the Primary Key

A `UNIQUE` constraint prevents duplicate values, separate from the primary key:

```
CREATE TABLE artists (
    artist_id bigserial CONSTRAINT artist_key PRIMARY KEY,
    artist_name varchar(200) NOT NULL,
    CONSTRAINT artist_name_unique UNIQUE (artist_name)
);
```

This prevents inserting two artists with the same name. Whether that is desirable depends on whether you believe there is only one "John Williams" in the music industry. (Spoiler: there are at least two famous ones.)

7.2 Composite UNIQUE Constraints

Album titles are not unique by themselves (many artists have a self-titled album). But artist + title should be:

```
CREATE TABLE albums (
    album_id bigserial CONSTRAINT album_key PRIMARY KEY,
    album_title varchar(200) NOT NULL,
    artist_id bigint REFERENCES artists (artist_id),
    CONSTRAINT album_artist_unique UNIQUE (album_title, artist_id)
);
```

Now two different artists can both have "Greatest Hits," but the same artist cannot have two albums with the same title.

7.3 UNIQUE vs PRIMARY KEY

Feature	PRIMARY KEY	UNIQUE
Uniqueness	Yes	Yes
Allows NULL	No	Yes (one NULL)
Per table	Only one	Multiple allowed
Creates index	Yes	Yes

8 NOT NULL Constraints

8.1 Requiring Values

`NOT NULL` prevents a column from containing NULL:

```
CREATE TABLE artists (
    artist_id bigserial,
    artist_name varchar(200) NOT NULL,
    CONSTRAINT artist_key PRIMARY KEY (artist_id)
);
```

An artist without a name is not an artist. It is a mystery.

8.2 When to Use NOT NULL

Always NOT NULL	Often Nullable
`artist_name`	`label` (indie releases)
`album_title`	`genre` (ambiguous)
`track_title`	`duration_min` (unknown)
Foreign keys (usually)	Notes, descriptions

> **Tip**
>
> Default to NOT NULL. Only allow NULLs when there is a legitimate reason for missing data. Future you will appreciate the strictness, even if present you finds it annoying.

8.3 Knowledge Check 4

Which constraint would BEST prevent someone from inserting an album with `release_year = 3099`?

A) PRIMARY KEY

B) FOREIGN KEY

C) CHECK

D) UNIQUE

Think about: **Why** can't the other constraint types handle this?

8.4 Answer: Knowledge Check 4

C) CHECK

CHECK validates that values meet a logical condition like `BETWEEN 1900 AND 2100`.

9 Modifying Tables with ALTER TABLE

9.1 Because Nobody Gets It Right the First Time

`ALTER TABLE` lets you modify constraints and columns after creation:

```
-- Remove a constraint
ALTER TABLE artists
    DROP CONSTRAINT artist_name_unique;

-- Add a constraint back
ALTER TABLE artists
    ADD CONSTRAINT artist_name_unique UNIQUE (artist_name);
```

9.2 NOT NULL: Different Syntax

NOT NULL is a column property, not a named constraint, so it uses different syntax:

```
-- Remove NOT NULL
ALTER TABLE albums
    ALTER COLUMN genre DROP NOT NULL;

-- Add NOT NULL back
ALTER TABLE albums
    ALTER COLUMN genre SET NOT NULL;
```

9.3 Common ALTER TABLE Operations

Operation	Syntax
Drop constraint	`ALTER TABLE t DROP CONSTRAINT c;`
Add constraint	`ALTER TABLE t ADD CONSTRAINT c ...;`
Drop NOT NULL	`ALTER TABLE t ALTER COLUMN col DROP NOT NULL;`
Set NOT NULL	`ALTER TABLE t ALTER COLUMN col SET NOT NULL;`
Add column	`ALTER TABLE t ADD COLUMN col type;`

Operation	Syntax
Drop column	`ALTER TABLE t DROP COLUMN col;`
Rename column	`ALTER TABLE t RENAME COLUMN old TO new;`
Rename table	`ALTER TABLE t RENAME TO new_name;`

This is your "oops" toolkit. Use it wisely.

9.4 ALTER TABLE in Practice

```sql
-- You realize albums need a label column you forgot
ALTER TABLE albums
    ADD COLUMN label varchar(100);

-- Genre should actually be required
ALTER TABLE albums
    ALTER COLUMN genre SET NOT NULL;

-- Rename a column for clarity
ALTER TABLE albums
    RENAME COLUMN duration TO duration_min;
```

9.5 Knowledge Check 5

You created a table and forgot to make `email` NOT NULL. Which command fixes this?

A) `ALTER TABLE users ADD CONSTRAINT email NOT NULL;`

B) `ALTER TABLE users ALTER COLUMN email SET NOT NULL;`

C) `UPDATE TABLE users SET email NOT NULL;`

D) `ALTER TABLE users MODIFY email NOT NULL;`

Think about: **Why** is NOT NULL different from other constraints syntactically?

9.6 Answer: Knowledge Check 5

B) `ALTER TABLE users ALTER COLUMN email SET NOT NULL;`

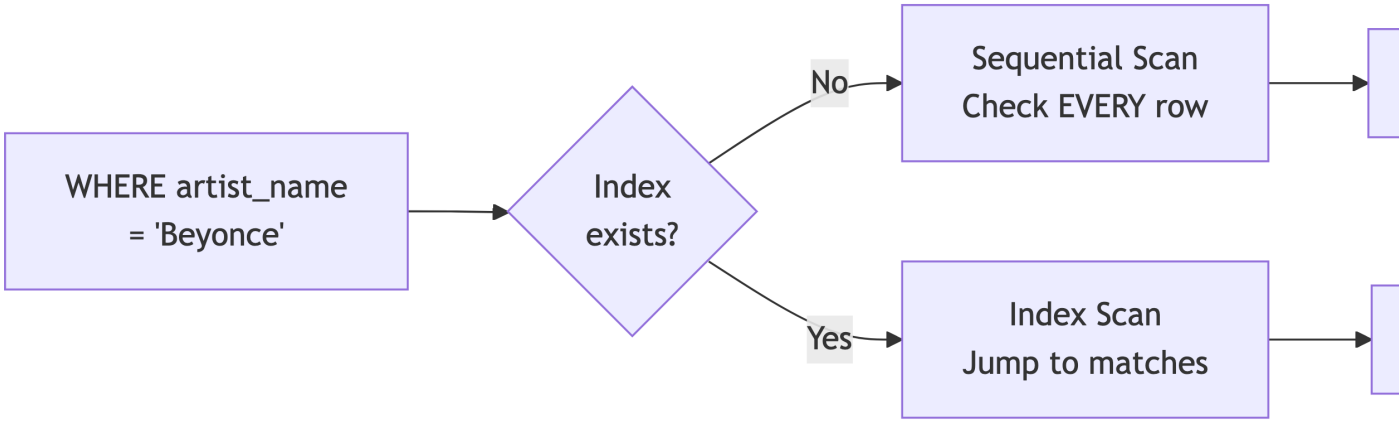
NOT NULL is a column property, not a named constraint, so it uses ALTER COLUMN.

10 Indexes: Making Queries Go Fast

10.1 What Is an Index?

An **index** is a data structure that speeds up retrieval at the cost of storage and slower writes.

Think of it like a textbook index: instead of reading every page to find "normalization," you look it up and jump to the right page. Databases without indexes are just very patient.



10.2 Creating Indexes

```
CREATE INDEX idx_albums_artist ON albums (artist_id);
CREATE INDEX idx_tracks_album ON tracks (album_id);
CREATE INDEX idx_albums_genre ON albums (genre);
```

These build **B-tree** indexes (the default) on frequently queried columns.

10.3 When to Index

Create Index When	Skip Index When
Column in WHERE clauses	Table is small (< 1000 rows)
Column in JOIN conditions	Column has few distinct values
Column in ORDER BY	Heavy INSERT/UPDATE load
Foreign key columns	You rarely query the column

> **Tip**
>
> PostgreSQL **automatically** indexes PRIMARY KEY and UNIQUE columns. You only need to create indexes on other frequently queried columns.

10.4 The Cost of Indexes

Indexes are not free:

- They consume disk space
- They slow down INSERT, UPDATE, and DELETE
- Too many indexes hurt overall performance

Indexing everything is like highlighting every word in a textbook. At that point, nothing is highlighted.

10.5 Knowledge Check 6

Which column would benefit MOST from an index?

A) A `notes` column that is never used in WHERE clauses

B) A `gender` column with only 3 possible values

C) An `email` column frequently used in WHERE and JOIN

D) A `created_at` column on a table with 50 rows

Think about: **Why** would indexes be wasteful on the other columns?

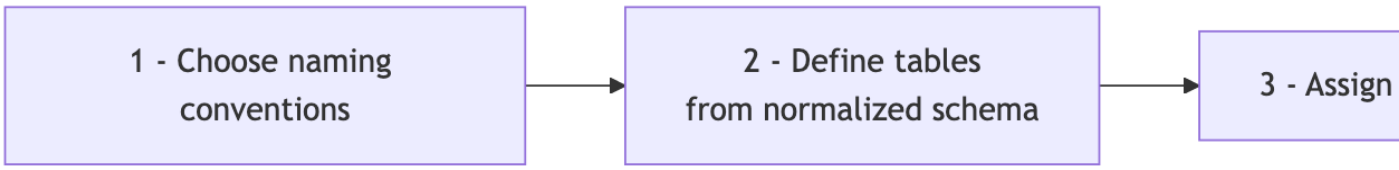
10.6 Answer: Knowledge Check 6

C) An `email` column frequently used in WHERE and JOIN

High selectivity + frequent queries = best index candidate.

11 Putting It All Together

11.1 The Build Order



11.2 The Artists Table

```
CREATE TABLE artists (
    artist_id bigserial,
    artist_name varchar(200) NOT NULL,
    CONSTRAINT artist_key PRIMARY KEY (artist_id),
    CONSTRAINT artist_name_unique UNIQUE (artist_name)
);
```

Decisions:

- Surrogate key because artist names can change
- `artist_name` is NOT NULL and UNIQUE

11.3 The Albums Table

```
CREATE TABLE albums (
    album_id bigserial,
    album_title varchar(200) NOT NULL,
    release_year smallint,
    genre varchar(50),
    label varchar(100),
    duration_min numeric(5,1),
    artist_id bigint NOT NULL REFERENCES artists (artist_id),
    CONSTRAINT album_key PRIMARY KEY (album_id),
    CONSTRAINT check_year_range
        CHECK (release_year BETWEEN 1900 AND 2100),
    CONSTRAINT check_duration_positive
        CHECK (duration_min > 0),
    CONSTRAINT album_artist_unique
```

```
        UNIQUE (album_title, artist_id)
);
```

Decisions:

- `artist_id` NOT NULL (every album needs an artist)
- CHECK on year catches obvious errors
- Composite UNIQUE prevents duplicate albums per artist

11.4 The Tracks Table

```
CREATE TABLE tracks (
    track_id bigserial,
    track_title varchar(200) NOT NULL,
    track_number smallint NOT NULL,
    duration_sec numeric(6,1),
    album_id bigint NOT NULL REFERENCES albums (album_id),
    CONSTRAINT track_key PRIMARY KEY (track_id),
    CONSTRAINT check_track_number CHECK (track_number > 0),
    CONSTRAINT check_track_duration CHECK (duration_sec > 0),
    CONSTRAINT track_album_unique UNIQUE (track_number, album_id)
);
```

Decisions:

- Composite UNIQUE on `(track_number, album_id)` prevents duplicate track numbers within an album
- `album_id` NOT NULL (orphan tracks are sad tracks)

11.5 The Indexes

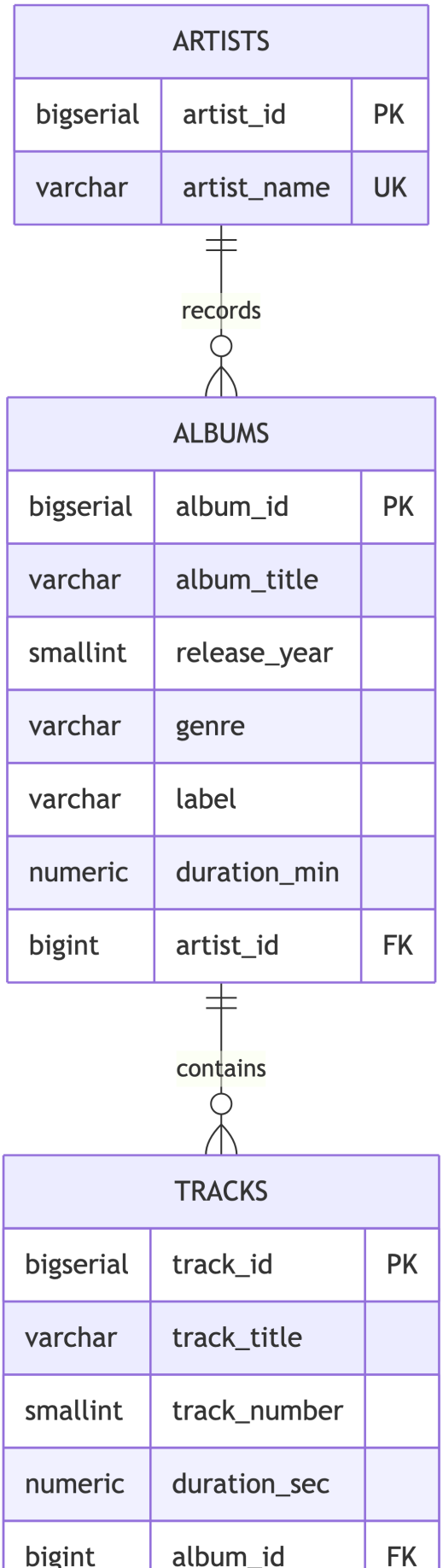
```
-- Speed up artist lookups on albums
CREATE INDEX idx_albums_artist ON albums (artist_id);

-- Speed up album lookups on tracks
CREATE INDEX idx_tracks_album ON tracks (album_id);

-- Speed up genre browsing
CREATE INDEX idx_albums_genre ON albums (genre);

-- Speed up year-based searches
CREATE INDEX idx_albums_year ON albums (release_year);
```

11.6 The Complete Schema



Three tables. Proper constraints. Indexes on the right columns. Ready for data.

11.7 Constraint Cheat Sheet

Constraint	Purpose	Syntax
PRIMARY KEY	Unique row identifier	`CONSTRAINT name PRIMARY KEY (col)`
FOREIGN KEY	Referential integrity	`col type REFERENCES table (col)`
CHECK	Value validation	`CONSTRAINT name CHECK (expr)`
UNIQUE	No duplicates	`CONSTRAINT name UNIQUE (col)`
NOT NULL	Requires a value	`col type NOT NULL`

12 What Is Next

12.1 Coming Up

Now that we can **build** tables, the next step is **filling** them. Next time:

- Audit the staging data for quality issues (spoiler: there are many)
- Fix inconsistencies with UPDATE
- Migrate data into our normalized tables with INSERT INTO … SELECT
- Wrap it all in transactions for safety

13 References

13.1 Sources

1. DeBarros, A. (2022). *Practical SQL: A Beginner's Guide to Storytelling with Data* (2nd ed.). No Starch Press. Chapter 7: Table Design That Works for You.

2. PostgreSQL Documentation. “CREATE TABLE.” https://www.postgresql.org/docs/current/sql-createtable.html

3. PostgreSQL Documentation. “Constraints.” https://www.postgresql.org/docs/current/ddl-constraints.html

4. PostgreSQL Documentation. “Indexes.” https://www.postgresql.org/docs/current/indexes.html