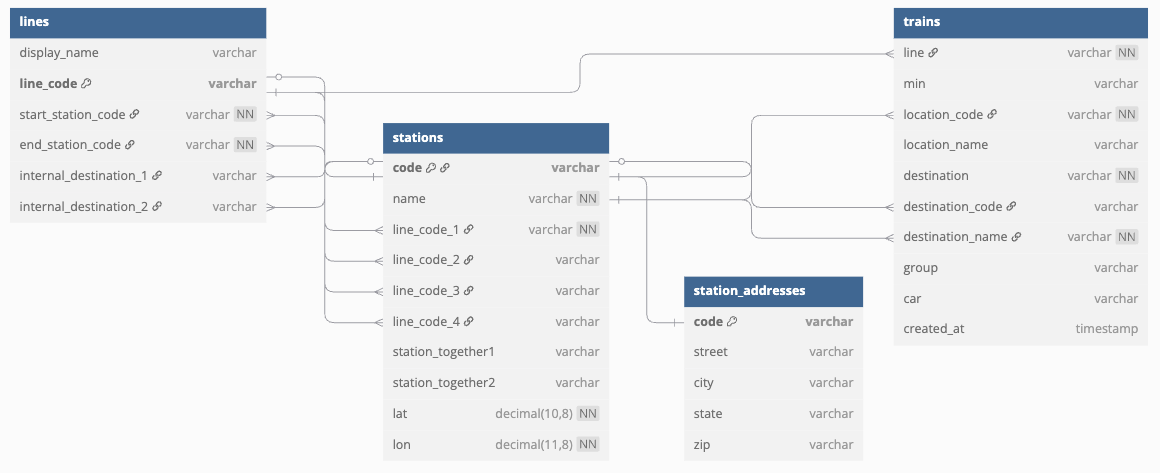
Metro Train Prediction App - Database Architecture Guide

Author: David Morrison

Project Repo: <https://github.com/DavMorr/dspolitical-wmata-app>



DBML: https://dbdiagram.io/d/WMAT-API-data-mapping-v2-684071dc76955641c2a87aa8

[Metro Train Prediction App Database Architecture Guide 1](#_Toc467171579)

[Overview 1](#_Toc1433342450)

[Database Design Philosophy 1](#_Toc243343937)

[Supported Use Cases 1](#_Toc1561770618)

[Database Schema 1](#_Toc1047947816)

[Entity Relationship Diagram 1](#_Toc2058538473)

[Table Summary 1](#_Toc48655287)

[Entity Relationships 1](#_Toc1760997256)

[Lines to Stations (Many-to-Many) 1](#_Toc1857728270)

[Lines to Station Paths (One-to-Many) 1](#_Toc1328272613)

[Stations to Station Addresses (One-to-One) 1](#_Toc85864430)

[Connected Stations (Self-Referencing) 1](#_Toc1665907389)

[Table Specifications 1](#_Toc372048959)

[lines 1](#_Toc219797852)

[stations 1](#_Toc1754208688)

[station\_addresses 1](#_Toc1265252975)

[station\_paths 1](#_Toc1081365844)

[Indexing Strategy 1](#_Toc33659913)

[Primary Indexes 1](#_Toc317838390)

[Performance Indexes 1](#_Toc238872439)

[stations Table 1](#_Toc533765539)

[station\_paths Table 1](#_Toc1136646996)

[station\_addresses Table 1](#_Toc1762520719)

[Index Usage Analysis 1](#_Toc1393536325)

[Query Patterns 1](#_Toc1344880216)

[Common Application Queries 1](#_Toc132408196)

[1. Get All Lines for Selection 1](#_Toc1065933964)

[2. Get Ordered Stations for Line 1](#_Toc1808079457)

[3. Get Stations Serving Multiple Lines (Transfer Stations) 1](#_Toc1647678194)

[4. Get Station with Address Information 1](#_Toc421997670)

[5. Find Nearby Stations (Geospatial) 1](#_Toc1966207963)

[Administrative Queries 1](#_Toc746325139)

[Data Sync Operations 1](#_Toc538455389)

[Cache Warming Queries 1](#_Toc1295026588)

[Performance Optimization 1](#_Toc2135683261)

[Query Performance Targets 1](#_Toc1848169387)

[Optimization Strategies 1](#_Toc112089552)

[1. Denormalization for Performance 1](#_Toc445073181)

[2. Composite Index Usage 1](#_Toc1982573408)

[3. String Primary Key Optimization 1](#_Toc180644820)

[4. Geospatial Query Optimization 1](#_Toc791450059)

[Database Configuration Recommendations 1](#_Toc248271777)

[MySQL Settings 1](#_Toc2050855680)

[Index Monitoring 1](#_Toc1225229380)

[Data Integrity 1](#_Toc1598717101)

[Referential Integrity Constraints 1](#_Toc793892008)

[Foreign Key Relationships 1](#_Toc159088105)

[Unique Constraints 1](#_Toc1701027498)

[Data Validation Rules 1](#_Toc85110810)

[Application-Level Validations 1](#_Toc663320534)

[Database Constraints 1](#_Toc659876191)

[Data Consistency Checks 1](#_Toc1691739518)

[Station-Line Relationship Validation 1](#_Toc1597144507)

[Sequence Consistency Validation 1](#_Toc653032268)

[Migration Strategy 1](#_Toc346017217)

[Migration Execution Order 1](#_Toc2063734531)

[Migration Files 1](#_Toc485862658)

[1. Create Lines Table 1](#_Toc161343909)

[2. Create Stations Table 1](#_Toc118111167)

[Rollback Strategy 1](#_Toc1150554406)

[Data Seeding Strategy 1](#_Toc921646528)

## Overview

### Database Design Philosophy

The Metro Train Prediction App database is designed around the real-world complexity of the Washington Metro system, emphasizing:

* **Natural Primary Keys** - Using actual Metro codes (line codes, station codes) as primary keys
* **Transfer Station Support** - Multi-line stations handled through flexible line code fields
* **Geographic Ordering** - Station sequences maintained for proper route display
* **High-Precision Geospatial Data** - Coordinates stored with sub-meter accuracy
* **Performance-First Indexing** - Indexes designed around actual query patterns

### Supported Use Cases

* Progressive form navigation (Lines → Stations → Predictions)
* Transfer station handling (stations serving multiple lines)
* Geographic route planning and distance calculations
* Real-time prediction display with station metadata
* Administrative data synchronization from WMATA API

## Database Schema

### Entity Relationship Diagram

LINES STATION\_PATHS

┌─────────────────────┐ ┌─────────────────────┐

│ line\_code (PK) │◄───────┤ line\_code (FK) │

│ display\_name │ │ station\_code (FK) │

│ start\_station\_code │ │ station\_name │

│ end\_station\_code │ │ seq\_num │

│ internal\_dest\_1 │ │ distance\_to\_prev │

│ internal\_dest\_2 │ │ id (PK) │

└─────────────────────┘ └─────────────────────┘

│ │

│ │

│ ▼

│ STATIONS

│ ┌─────────────────────┐

└─────────────────────►│ code (PK) │

│ name │

│ line\_code\_1 │

│ line\_code\_2 │

│ line\_code\_3 │

│ line\_code\_4 │

│ station\_together\_1 │

│ station\_together\_2 │

│ lat │

│ lon │

│ is\_active │

└─────────────────────┘

│

│ 1:1

▼

STATION\_ADDRESSES

┌─────────────────────┐

│ station\_code (PK) │

│ street │

│ city │

│ state │

│ zip\_code │

│ country │

└─────────────────────┘

### Table Summary

|  |  |  |  |
| --- | --- | --- | --- |
| **Table** | **Purpose** | **Primary Key** | **Records** |
| lines | Metro line definitions | line\_code | ~6 |
| stations | Station master data | code | ~95 |
| station\_addresses | Station location details | station\_code | ~95 |
| station\_paths | Geographic station ordering | id | ~95 |

## Entity Relationships

### Lines to Stations (Many-to-Many)

Relationship: A line serves multiple stations; a station can serve multiple lines

Implementation: Through line\_code\_1, line\_code\_2, line\_code\_3, line\_code\_4 fields in stations table

Examples:

sql

*-- Red Line serves stations A15 through B11*

SELECT \* FROM stations WHERE line\_code\_1 = 'RD' OR line\_code\_2 = 'RD';

*-- Metro Center (A01) serves Red, Blue, Orange, Silver lines*

SELECT \* FROM stations WHERE code = 'A01';

*-- Result: line\_code\_1='RD', line\_code\_2='BL', line\_code\_3='OR', line\_code\_4='SV'*

### Lines to Station Paths (One-to-Many)

Relationship: Each line has an ordered sequence of stations

Implementation: station\_paths table with line\_code foreign key and seq\_num ordering

Purpose: Provides geographic ordering for station lists in the frontend

### Stations to Station Addresses (One-to-One)

Relationship: Each station has exactly one address

Implementation: station\_addresses table with station\_code primary key matching stations.code

Cascade Behavior: Deleting a station removes its address

### Connected Stations (Self-Referencing)

Relationship: Stations can reference other station platforms

Implementation: station\_together\_1 and station\_together\_2 fields

Example: Metro Center has platforms A01 and C01 (connected but separate codes)

## Table Specifications

### **lines**

Purpose: Metro line definitions and termination points

sql

CREATE TABLE lines (

line\_code VARCHAR(2) PRIMARY KEY, *-- RD, BL, GR, OR, SV, YL*

display\_name VARCHAR(50) NOT NULL, *-- Red, Blue, Green, etc.*

start\_station\_code VARCHAR(3) NOT NULL, *-- A15 (Shady Grove)*

end\_station\_code VARCHAR(3) NOT NULL, *-- B11 (Glenmont)*

internal\_destination\_1 VARCHAR(3), *-- Branch/split destinations*

internal\_destination\_2 VARCHAR(3),

created\_at TIMESTAMP,

updated\_at TIMESTAMP,

INDEX idx\_display\_name (display\_name)

);

Key Features:

* Natural primary key using WMATA line codes
* References to start/end stations for route definition
* Internal destinations for handling route branches
* Display name index for alphabetical sorting

Sample Data:

sql

INSERT INTO lines VALUES

('RD', 'Red', 'A15', 'B11', NULL, NULL),

('BL', 'Blue', 'J03', 'G05', NULL, NULL),

('GR', 'Green', 'E10', 'F11', 'C15', 'D13');

### **stations**

Purpose: Master station data with multi-line support

sql

CREATE TABLE stations (

code VARCHAR(3) PRIMARY KEY, *-- A01, B02, C03, etc.*

name VARCHAR(100) NOT NULL, *-- Metro Center, Union Station*

line\_code\_1 VARCHAR(2), *-- Primary line*

line\_code\_2 VARCHAR(2), *-- Transfer line 1*

line\_code\_3 VARCHAR(2), *-- Transfer line 2*

line\_code\_4 VARCHAR(2), *-- Transfer line 3*

station\_together\_1 VARCHAR(3), *-- Connected platform 1*

station\_together\_2 VARCHAR(3), *-- Connected platform 2*

lat DECIMAL(10,8) NOT NULL, *-- 38.89834567*

lon DECIMAL(11,8) NOT NULL, *-- -77.02834567*

is\_active BOOLEAN DEFAULT true,

created\_at TIMESTAMP,

updated\_at TIMESTAMP,

INDEX idx\_name (name),

INDEX idx\_coordinates (lat, lon),

INDEX idx\_is\_active (is\_active)

);

Key Features:

* 3-character station codes as natural primary keys
* Up to 4 line codes for complex transfer stations
* High-precision coordinates (11mm accuracy)
* Connected station references for platform relationships
* Active status for filtering operational stations

Sample Data:

sql

INSERT INTO stations VALUES

('A01', 'Metro Center', 'RD', 'BL', 'OR', 'SV', 'C01', NULL, 38.89834567, -77.02834567, true),

('A02', 'Farragut North', 'RD', NULL, NULL, NULL, NULL, NULL, 38.90344123, -77.03927456, true),

('L01', 'Gallery Pl-Chinatown', 'RD', 'YL', 'GR', NULL, 'B01', 'F01', 38.89766789, -77.02112345, true);

### **station\_addresses**

Purpose: Physical address information for stations

sql

CREATE TABLE station\_addresses (

station\_code VARCHAR(3) PRIMARY KEY, *-- Matches stations.code*

street VARCHAR(255) NOT NULL, *-- 1001 G St NW*

city VARCHAR(100) NOT NULL, *-- Washington*

state VARCHAR(2) NOT NULL, *-- DC*

zip\_code VARCHAR(10) NOT NULL, *-- 20001-1234*

country VARCHAR(2) DEFAULT 'US',

created\_at TIMESTAMP,

updated\_at TIMESTAMP,

FOREIGN KEY (station\_code) REFERENCES stations(code) ON DELETE CASCADE,

INDEX idx\_city\_state (city, state),

INDEX idx\_zip\_code (zip\_code)

);

Key Features:

* One-to-one relationship with stations
* Cascade delete maintains referential integrity
* Indexes for location-based searches
* Supports international addresses (country field)

### **station\_paths**

Purpose: Geographic ordering of stations along metro lines

sql

CREATE TABLE station\_paths (

id BIGINT PRIMARY KEY AUTO\_INCREMENT,

line\_code VARCHAR(2) NOT NULL, *-- RD, BL, etc.*

station\_code VARCHAR(3) NOT NULL, *-- A01, B02, etc.*

station\_name VARCHAR(100) NOT NULL, *-- Duplicated for performance*

seq\_num INTEGER NOT NULL, *-- 1, 2, 3... (geographic order)*

distance\_to\_prev INTEGER DEFAULT 0, *-- Meters to previous station*

created\_at TIMESTAMP,

updated\_at TIMESTAMP,

FOREIGN KEY (line\_code) REFERENCES lines(line\_code),

FOREIGN KEY (station\_code) REFERENCES stations(code),

INDEX idx\_line\_sequence (line\_code, seq\_num),

INDEX idx\_station\_code (station\_code),

UNIQUE KEY uk\_line\_station (line\_code, station\_code)

);

Key Features:

* Surrogate primary key for flexibility
* Composite foreign keys to both lines and stations
* Sequential numbering for geographic ordering
* Distance tracking for travel time calculations
* Unique constraint prevents duplicate entries

Sample Data:

sql

INSERT INTO station\_paths VALUES

(1, 'RD', 'A15', 'Shady Grove', 1, 0),

(2, 'RD', 'A14', 'Rockville', 2, 4823),

(3, 'RD', 'A13', 'Twinbrook', 3, 2134),

(4, 'RD', 'A12', 'White Flint', 4, 1876);

## Indexing Strategy

### **Primary Indexes**

All tables use optimized primary keys:

* lines.line\_code - String primary key (2 chars)
* stations.code - String primary key (3 chars)
* station\_addresses.station\_code - String primary key (3 chars)
* station\_paths.id - Auto-increment integer

### Performance Indexes

#### **stations Table**

sql

INDEX idx\_name (name) *-- Station name searches/sorting*

INDEX idx\_coordinates (lat, lon) *-- Geospatial queries*

INDEX idx\_is\_active (is\_active) *-- Filter active stations*

#### **station\_paths Table**

sql

INDEX idx\_line\_sequence (line\_code, seq\_num) *-- Ordered station retrieval*

INDEX idx\_station\_code (station\_code) *-- Individual station lookup*

UNIQUE KEY uk\_line\_station (line\_code, station\_code) *-- Prevent duplicates*

#### **station\_addresses Table**

sql

INDEX idx\_city\_state (city, state) *-- Location-based searches*

INDEX idx\_zip\_code (zip\_code) *-- Postal code lookups*

### **Index Usage Analysis**

sql

*-- This query uses idx\_line\_sequence for optimal performance*

SELECT \* FROM station\_paths

WHERE line\_code = 'RD'

ORDER BY seq\_num;

*-- This query uses idx\_coordinates for geospatial searches*

SELECT \* FROM stations

WHERE lat BETWEEN 38.89 AND 38.91

AND lon BETWEEN -77.04 AND -77.02;

*-- Multi-line station query (no specific index, but fast due to small table)*

SELECT \* FROM stations

WHERE line\_code\_1 = 'RD' OR line\_code\_2 = 'RD'

OR line\_code\_3 = 'RD' OR line\_code\_4 = 'RD';

## Query Patterns

### **Common Application Queries**

#### **1. Get All Lines for Selection**

sql

SELECT line\_code as value, display\_name as label

FROM lines

ORDER BY display\_name;

Performance: ~1ms (6 rows, primary key scan)

#### **2. Get Ordered Stations for Line**

sql

SELECT sp.station\_code as value, sp.station\_name as label,

sp.seq\_num, sp.distance\_to\_prev

FROM station\_paths sp

WHERE sp.line\_code = ?

ORDER BY sp.seq\_num;

Performance: ~5ms (uses idx\_line\_sequence)

#### **3. Get Stations Serving Multiple Lines (Transfer Stations)**

sql

SELECT s.code, s.name, s.line\_code\_1, s.line\_code\_2, s.line\_code\_3, s.line\_code\_4

FROM stations s

WHERE s.line\_code\_1 = ? OR s.line\_code\_2 = ?

OR s.line\_code\_3 = ? OR s.line\_code\_4 = ?;

Performance: ~10ms (table scan, but small table)

#### **4. Get Station with Address Information**

sql

SELECT s.code, s.name, s.lat, s.lon,

sa.street, sa.city, sa.state, sa.zip\_code

FROM stations s

LEFT JOIN station\_addresses sa ON s.code = sa.station\_code

WHERE s.code = ?;

Performance: ~2ms (primary key lookup + join)

#### **5. Find Nearby Stations (Geospatial)**

sql

SELECT code, name, lat, lon,

(6371000 \* acos(cos(radians(?)) \* cos(radians(lat)) \*

cos(radians(lon) - radians(?)) + sin(radians(?)) \*

sin(radians(lat)))) AS distance

FROM stations

WHERE lat BETWEEN ? - 0.01 AND ? + 0.01

AND lon BETWEEN ? - 0.01 AND ? + 0.01

AND is\_active = true

ORDER BY distance

LIMIT 10;

Performance: ~15ms (uses idx\_coordinates for initial filtering)

### Administrative Queries

#### **Data Sync Operations**

sql

*-- Insert/Update Lines*

INSERT INTO lines (line\_code, display\_name, start\_station\_code, end\_station\_code)

VALUES (?, ?, ?, ?)

ON DUPLICATE KEY UPDATE

display\_name = VALUES(display\_name),

start\_station\_code = VALUES(start\_station\_code),

end\_station\_code = VALUES(end\_station\_code);

*-- Clear and Rebuild Paths*

DELETE FROM station\_paths WHERE line\_code = ?;

INSERT INTO station\_paths (line\_code, station\_code, station\_name, seq\_num, distance\_to\_prev)

VALUES (?, ?, ?, ?, ?);

#### **Cache Warming Queries**

sql

*-- Preload all station-line relationships*

SELECT s.code, s.name, s.line\_code\_1, s.line\_code\_2, s.line\_code\_3, s.line\_code\_4

FROM stations s

WHERE s.is\_active = true;

*-- Preload all line paths*

SELECT line\_code, station\_code, station\_name, seq\_num

FROM station\_paths

ORDER BY line\_code, seq\_num;

## Performance Optimization

### Query Performance Targets

* **Line selection**: < 5ms
* **Station list for line**: < 10ms
* **Station details**: < 5ms
* **Geospatial searches**: < 20ms
* **Multi-line station queries**: < 15ms

### Optimization Strategies

#### **1. Denormalization for Performance**

The station\_paths table duplicates station names for performance:

sql

*-- Instead of always joining to stations table*

SELECT sp.station\_code, s.name

FROM station\_paths sp

JOIN stations s ON sp.station\_code = s.code

WHERE sp.line\_code = 'RD';

*-- We can query directly (faster)*

SELECT station\_code, station\_name

FROM station\_paths

WHERE line\_code = 'RD';

#### **2. Composite Index Usage**

The (line\_code, seq\_num) index optimizes the most common query pattern:

sql

EXPLAIN SELECT \* FROM station\_paths WHERE line\_code = 'RD' ORDER BY seq\_num;

*-- Uses: idx\_line\_sequence (covering index)*

*-- Rows examined: ~27 (only Red Line stations)*

*-- Extra: Using index*

#### **3. String Primary Key Optimization**

Using actual Metro codes as primary keys provides benefits:

* **Meaningful joins**: WHERE line\_code = 'RD' is more readable than WHERE line\_id = 1
* **Cache efficiency**: Natural keys are more cache-friendly
* **Reduced joins**: No need to join for display values

#### **4. Geospatial Query Optimization**

For location-based queries, use bounding box filtering before distance calculations:

sql

*-- Efficient: Filter with index first, then calculate distance*

SELECT \*, (complex\_distance\_calculation) as distance

FROM stations

WHERE lat BETWEEN ? AND ? *-- Uses index for initial filtering*

AND lon BETWEEN ? AND ? *-- Further reduces candidate set*

AND is\_active = true

ORDER BY distance;

### Database Configuration Recommendations

#### **MySQL Settings**

sql

*-- Optimize for read-heavy workload*

innodb\_buffer\_pool\_size = 1G *-- Cache frequently accessed data*

query\_cache\_size = 128M *-- Cache repeated queries*

query\_cache\_type = 1 *-- Enable query cache*

*-- Index optimization*

innodb\_stats\_on\_metadata = 0 *-- Reduce metadata overhead*

optimizer\_search\_depth = 4 *-- Optimize join planning*

#### **Index Monitoring**

sql

*-- Check index usage*

SELECT

s.table\_name,

s.index\_name,

s.cardinality,

round(((s.cardinality / t.table\_rows) \* 100), 2) as selectivity

FROM information\_schema.statistics s

INNER JOIN information\_schema.tables t

ON s.table\_schema = t.table\_schema

AND s.table\_name = t.table\_name

WHERE s.table\_schema = 'metro\_transit'

ORDER BY s.table\_name, selectivity;

## Data Integrity

### Referential Integrity Constraints

#### **Foreign Key Relationships**

sql

*-- Station addresses must reference valid stations*

ALTER TABLE station\_addresses

ADD CONSTRAINT fk\_station\_addresses\_station\_code

FOREIGN KEY (station\_code) REFERENCES stations(code) ON DELETE CASCADE;

*-- Station paths must reference valid lines and stations*

ALTER TABLE station\_paths

ADD CONSTRAINT fk\_station\_paths\_line\_code

FOREIGN KEY (line\_code) REFERENCES lines(line\_code);

ALTER TABLE station\_paths

ADD CONSTRAINT fk\_station\_paths\_station\_code

FOREIGN KEY (station\_code) REFERENCES stations(code);

#### **Unique Constraints**

sql

*-- Prevent duplicate station entries per line*

ALTER TABLE station\_paths

ADD CONSTRAINT uk\_line\_station UNIQUE (line\_code, station\_code);

### Data Validation Rules

#### **Application-Level Validations**

php

*// Line codes must be 2 uppercase letters*

'line\_code' => 'required|regex:/^[A-Z]{2}$/'

*// Station codes must be 3 alphanumeric characters*

'code' => 'required|regex:/^[A-Z0-9]{3}$/'

*// Coordinates must be within reasonable bounds*

'lat' => 'required|numeric|between:38.0,39.5' *// DC area bounds*

'lon' => 'required|numeric|between:-78.0,-76.0'

*// Sequence numbers must be positive*

'seq\_num' => 'required|integer|min:1'

#### **Database Constraints**

sql

*-- Coordinate bounds checking*

ALTER TABLE stations

ADD CONSTRAINT chk\_lat\_bounds CHECK (lat BETWEEN 38.0 AND 39.5);

ALTER TABLE stations

ADD CONSTRAINT chk\_lon\_bounds CHECK (lon BETWEEN -78.0 AND -76.0);

*-- Sequence numbers must be positive*

ALTER TABLE station\_paths

ADD CONSTRAINT chk\_seq\_num\_positive CHECK (seq\_num > 0);

*-- Distance must be non-negative*

ALTER TABLE station\_paths

ADD CONSTRAINT chk\_distance\_non\_negative CHECK (distance\_to\_prev >= 0);

### Data Consistency Checks

#### **Station-Line Relationship Validation**

sql

*-- Find stations without any line assignments*

SELECT code, name

FROM stations

WHERE line\_code\_1 IS NULL

AND line\_code\_2 IS NULL

AND line\_code\_3 IS NULL

AND line\_code\_4 IS NULL;

*-- Find paths referencing non-existent stations*

SELECT sp.line\_code, sp.station\_code

FROM station\_paths sp

LEFT JOIN stations s ON sp.station\_code = s.code

WHERE s.code IS NULL;

#### **Sequence Consistency Validation**

sql

*-- Check for missing sequence numbers*

SELECT line\_code,

COUNT(\*) as station\_count,

MAX(seq\_num) as max\_seq,

MIN(seq\_num) as min\_seq

FROM station\_paths

GROUP BY line\_code

HAVING station\_count != (max\_seq - min\_seq + 1);

*-- Find duplicate sequence numbers within lines*

SELECT line\_code, seq\_num, COUNT(\*)

FROM station\_paths

GROUP BY line\_code, seq\_num

HAVING COUNT(\*) > 1;

## Migration Strategy

### Migration Execution Order

1. **Create base tables** (lines, stations)
2. **Create dependent tables** (station\_addresses, station\_paths)
3. **Add foreign key constraints**
4. **Create indexes**
5. **Load initial data**

### Migration Files

#### **1. Create Lines Table**

php

*// 2025\_06\_04\_184151\_create\_lines\_table.php*

Schema::create('lines', function (Blueprint $table) {

$table->string('line\_code', 2)->primary();

$table->string('display\_name', 50);

$table->string('start\_station\_code', 3);

$table->string('end\_station\_code', 3);

$table->string('internal\_destination\_1', 3)->nullable();

$table->string('internal\_destination\_2', 3)->nullable();

$table->timestamps();

$table->index('display\_name');

});

#### **2. Create Stations Table**

php

*// 2025\_06\_04\_184201\_create\_stations\_table.php*

Schema::create('stations', function (Blueprint $table) {

$table->string('code', 3)->primary();

$table->string('name', 100);

$table->string('line\_code\_1', 2)->nullable();

$table->string('line\_code\_2', 2)->nullable();

$table->string('line\_code\_3', 2)->nullable();

$table->string('line\_code\_4', 2)->nullable();

$table->string('station\_together\_1', 3)->nullable();

$table->string('station\_together\_2', 3)->nullable();

$table->decimal('lat', 10, 8);

$table->decimal('lon', 11, 8);

$table->boolean('is\_active')->default(true);

$table->timestamps();

$table->index('name');

$table->index(['lat', 'lon']);

$table->index('is\_active');

});

### Rollback Strategy

php

*// All migrations include proper down() methods*

public function down(): void

{

Schema::dropIfExists('station\_paths'); *// Drop dependent tables first*

Schema::dropIfExists('station\_addresses');

Schema::dropIfExists('stations');

Schema::dropIfExists('lines'); *// Drop parent tables last*

}

### Data Seeding Strategy

php  
*// DatabaseSeeder.php*

public function run(): void {  
 *// Metro Train Prediction App uses a custom sync command*   
 *// that pulls data directly from the WMATA API*   
 try {   
 $this->command->info('Syncing Metro data from WMATA API...');  
 $exitCode = Artisan::call('metro:sync');   
 if ($exitCode === 0) {   
 $this->command->info('Metro data sync completed successfully');   
 }   
 else {   
 $this->command->error('Metro data sync failed with exit code: ' . $exitCode);   
 throw new \Exception('Metro sync command failed');   
 }   
 }   
 catch (\Exception $e) {   
 $this->command->error('Failed to sync Metro data: ' . $e->getMessage());   
 $this->command->warn('Database seeding will continue, but Metro tables may be empty');  
 $this->command->warn('Run "sail artisan metro:sync" manually after resolving API issues');   
 *// Don't throw - allow other seeders to run*  
 *// throw $e; // Uncomment to fail entire seeding process*  
 }   
}

### Alternative Seeding Methods

bash  
*# Direct command execution (recommended)*  
sail artisan metro:sync

*# With validation check first*  
sail artisan metro:sync --validate

*# Via DatabaseSeeder during fresh installation*  
sail artisan db:seed

The Metro Train Prediction App uses a specialized sync command rather than traditional Laravel seeders because:

* Data is sourced from the live WMATA API
* Station relationships and paths are calculated dynamically
* Ensures data consistency with the current Metro system
* Handles complex multi-line station assignments automatically

This database architecture provides a robust foundation for the Metro Train Prediction App, optimized for the specific query patterns of the frontend while maintaining data integrity and supporting future expansion.