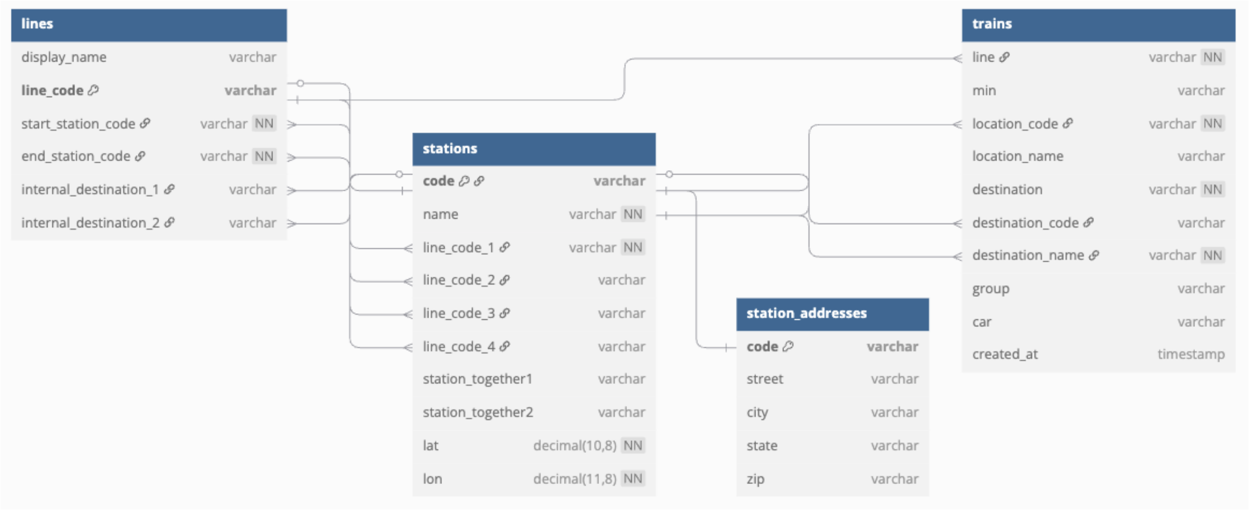
Metro Train Prediction App - Database Architecture Guide

Author: David Morrison

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



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

Overview

Database Design Philosophy

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

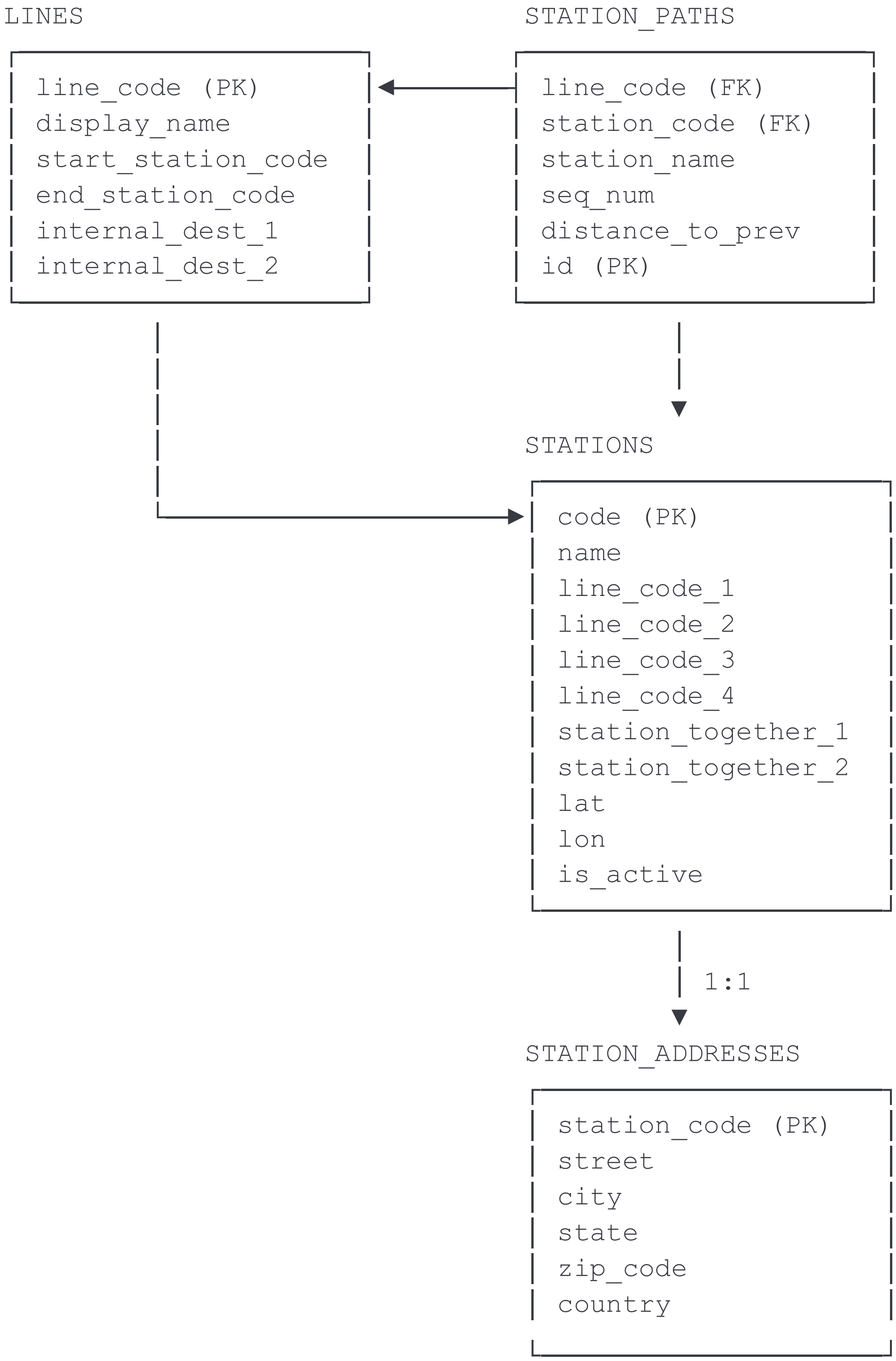
- **Natural Primary Keys** - Using actual Metro codes (line codes, station codes) as primary keys
- **Transfer Station Support** - Multi-line stations handled through flexible line code fields
- **Geographic Ordering** - Station sequences maintained for proper route display
- **High-Precision Geospatial 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

line_code (PK)
display_name
start_station_code
end_station_code
internal_dest_1
internal_dest_2

STATION_PATHS

line_code (FK)
station_code (FK)
station_name
seq_num
distance_to_prev
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

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
```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

```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

```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

```sql
INDEX idx_city_state (city, state)       -- Location-based searches
INDEX idx_zip_code (zip_code)            -- Postal code lookups
```

Index Usage Analysis

sql

```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

```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

```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

```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

```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

```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

```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.