Schema Migration Strategies

1. The Real Problem This Exists to Solve

Database schemas evolve as products grow. Adding columns, renaming fields, changing data types, or restructuring tables is inevitable. But production databases contain live data serving active users. Schema changes must happen without downtime, data loss, or breaking running application code.

Real production scenario:

- Rename column `user.name` to `user.full_name`
- 10 million rows in users table
- 100 application servers reading/writing to database
- **Naive approach (immediate rename):**

  ```
  ALTER TABLE users RENAME COLUMN name TO full_name;
  ```

  - Rename executes instantly
  - Old application code: `SELECT name FROM users`
  - Database error: "column 'name' does not exist"
  - All 100 servers throw errors
  - Complete outage: 0% success rate
  - Rollback requires another rename (more downtime)
  - Users unable to login, make purchases, access data
  - Revenue lost, reputation damaged
- **Correct approach (expand-migrate-contract):**

  1. **Expand:** Add `full_name` column, keep `name` (backward compatible)
  2. **Migrate data:** Copy `name` → `full_name`
  3. **Deploy application:** Update code to write both columns, read from `full_name`
  4. **Backfill:** Ensure all rows have `full_name` populated
  5. **Contract:** Remove `name` column (all code updated)

  - Zero downtime
  - No errors
  - Gradual migration
  - Rollback safe (old column exists during transition)

The fundamental problem: Application code and database schema are tightly coupled. Changing the schema breaks running code unless you carefully manage the transition. Deployments must be coordinated with schema changes, and both must support the old and new schemas simultaneously during the migration period.

Without proper migration strategy:

- Downtime required for schema changes
- Application errors during migration
- Risk of data loss
- Impossible to rollback safely

- Coordination nightmare (stop all servers, migrate, restart)

With proper migration strategy:

- Zero downtime deployments
- Backward and forward compatibility
- Safe rollback at any point
- Gradual data migration
- Independent deployment of code and schema

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Direct Schema Change (Breaking Change)

```
-- Incorrect: Rename column directly
ALTER TABLE users RENAME COLUMN email TO email_address;

-- Immediately breaks all application code:
SELECT email FROM users; -- ERROR: column "email" does not exist
```

Why it seems reasonable:

- Simple one-line change
- Database updated instantly
- Reflects desired final state

How it breaks:

```
Production:
- 50 application servers running
- Each server executes: SELECT email FROM users
- Schema changed: email → email_address
- All queries fail: "column 'email' does not exist"
- Error rate: 100%
- Complete outage

Timeline:
14:00 - Execute ALTER TABLE
14:00:01 - All servers start throwing errors
14:00:05 - Monitoring alerts: 100% error rate
14:00:10 - Emergency response team assembled
14:00:15 - Decision to rollback
14:00:20 - Execute: ALTER TABLE users RENAME COLUMN email_address TO email
14:00:21 - Service restored

Total downtime: 21 seconds
User impact: Every user unable to access service
```

Production symptoms:

```
Application logs:
Error: column "email" does not exist
  at SELECT email FROM users WHERE id = $1
```

```
  [repeated 100,000 times]

Database CPU: 5% (barely working, all queries failing)
Application error rate: 100%
Load balancer health checks: All failing
```

❌ Incorrect Approach #2: Change Schema Then Deploy Code (Race Condition)

```sql
-- Step 1: Add new column
ALTER TABLE users ADD COLUMN email_address VARCHAR(255);

-- Step 2: Deploy application code (reads email_address)
-- Problem: NULL values for all existing rows!
```

Why it seems reasonable:

- Additive change (backward compatible)
- New column doesn't break old queries
- Progressive migration

How it breaks:

```
Timeline:
14:00 - Add email_address column (all rows have NULL)
14:05 - Start deploying new application code
14:05 - Server 1 deployed (reads email_address)
14:10 - Server 10 deployed
14:15 - Server 20 deployed
...
14:45 - All 50 servers deployed

Problem during deployment (14:05 - 14:45):
- New code: SELECT email_address FROM users WHERE id = 123
- Returns: NULL (data not migrated yet!)
- Application logic: if (!user.emailAddress) throw error
- User cannot login: "Email address required"

Impact:
- 40 minutes of partial outage
- Users see: "Email address missing" errors
- Some servers work (old code), some don't (new code)
- Inconsistent user experience
```

Production symptoms:

```
User reports:
"I can't login, it says my email is missing!"
"It worked 5 minutes ago..."
"Some pages work, some don't."

Database:
email_address column: [NULL, NULL, NULL, ...] (not migrated)
```

```
Application logs:
Server 1-10: Reading email_address, getting NULL
Server 11-50: Reading email, working fine
Inconsistent behavior across fleet
```

❌ Incorrect Approach #3: Backfill During Peak Traffic (Performance Killer)

```sql
-- Incorrect: Migrate 10 million rows during business hours
UPDATE users SET full_name = name WHERE full_name IS NULL;

-- Locks entire table, blocks reads/writes
```

Why it seems reasonable:

- Single query to migrate data
- Seems fast and simple
- Gets migration done

How it breaks:

```
Database:
- Users table: 10 million rows
- UPDATE locks entire table (row-level locks × 10M)
- Other queries: SELECT, INSERT, UPDATE blocked
- Migration time: 5 minutes (large table)

Timeline:
15:00 - Execute UPDATE users SET full_name = name
15:00 - Table locked
15:00:01 - Incoming queries start queuing
15:00:05 - 1000 queries waiting
15:00:10 - 5000 queries waiting
15:00:30 - 20,000 queries waiting
15:01:00 - Connection pool exhausted
15:02:00 - Load balancer: All servers unhealthy
15:05:00 - UPDATE completes, table unlocked

Impact:
- 5 minutes of degraded performance
- Queries timing out
- Connection pool exhaustion
- Users seeing "Database unavailable" errors
- Monitoring: Database CPU 100%, slow queries
```

Production symptoms:

```
Database metrics:
- Active queries: 20,000 (normally 50)
- Lock wait time: 5 minutes
- CPU: 100%
- Disk I/O: 100%
```

```
Application:
- Connection pool: 0 available (all blocked)
- Timeouts: 90% of requests
- Error rate: 80%

User experience:
- "Your request is taking longer than expected..."
- Page load times: 60+ seconds
- Checkout failures
- Cart abandonments
```

❌ Incorrect Approach #4: No Rollback Plan (One-Way Door)

```sql
-- Incorrect: Drop old column immediately after code deployment
ALTER TABLE users ADD COLUMN email_address VARCHAR(255);
UPDATE users SET email_address = email;
-- Deploy new code
ALTER TABLE users DROP COLUMN email; -- DANGER!

-- If new code has bug, can't rollback (email column gone)
```

Why it seems reasonable:

- Clean migration
- Remove deprecated fields
- Forward progress

How it breaks:

```
Migration steps:
1. Add email_address column
2. Copy data: email → email_address
3. Deploy new code (uses email_address)
4. Drop email column

Scenario:
- Step 4 executed: email column dropped
- New code deployed: reads email_address
- Bug discovered: email_address has data corruption
- Need to rollback to old code
- Old code expects email column
- email column doesn't exist!
- Rollback fails

Result:
- Can't move forward (new code buggy)
- Can't move backward (old column gone)
- Stuck in broken state
- Requires manual intervention:
  * Restore from backup
```

```
  * Re-add email column
  * Re-migrate data
  * Downtime required
```

Production symptoms:

```
Incident:
16:00 - Drop email column
16:10 - Bug discovered in new code
16:11 - Attempt rollback to old code
16:12 - Old code crashes: "column 'email' does not exist"
16:13 - Both old and new code broken
16:15 - Emergency database restoration
16:45 - Service restored from backup

Total downtime: 45 minutes
Root cause: No rollback plan
Lesson: Never drop columns immediately after deployment
```

❌ Incorrect Approach #5: No Migration Versioning (Applied Twice)

```sql
-- migrations/001_add_email.sql
ALTER TABLE users ADD COLUMN email VARCHAR(255);

-- Run manually:
psql -f migrations/001_add_email.sql

-- Problem: Run twice by accident
-- Error: column "email" already exists
```

Why it seems reasonable:

- SQL files for each migration
- Manual execution
- Simple to understand

How it breaks:

```
Scenario 1: Accidental re-run
Developer 1: Runs migration on staging
Developer 2: Doesn't know, runs same migration
Result: Error, staging database broken

Scenario 2: Missing migration
Developer: Skips migration 003, runs 004
Result: Dependency missing, 004 fails

Scenario 3: Drift between environments
Production: Migrations 001, 002, 003
Staging: Migrations 001, 003 (002 skipped)
Result: Environments out of sync, bugs appear in production
```

```
Scenario 4: No audit trail
Question: "Which migrations ran on production?"
Answer: "Uh... we think all of them?"
Result: Uncertainty, risk of data loss
```

Production symptoms:

```
Deployment:
Production engineer: "Did migration 023 run?"
Team: "I think so?"
Engineer: "Let me run it to be safe..."
Result: ERROR - duplicate column

Actual state:
Production: [001-020, 022, 023] (021 missing!)
Staging: [001-023]
No source of truth
Manual detective work required
```

3. Correct Mental Model (How It Actually Works)

Schema migrations follow the **expand-migrate-contract** pattern:

1. **Expand:** Add new schema elements (additive, backward compatible)
2. **Migrate:** Dual-write to old and new schema, backfill existing data
3. **Contract:** Remove old schema elements (after all code updated)

Each phase is a separate deployment, with waiting periods between phases to ensure safety.

Expand Phase

```
Old schema:
users: [id, name, email]

Expand (add new column):
users: [id, name, email, email_address]

Old code still works:
SELECT name, email FROM users ✓

New code can use new column:
SELECT name, email_address FROM users ✓ (NULL for now)
```

Migrate Phase

```
Dual write (application code):
user.email = newValue;
user.email_address = newValue; // Write to both

Backfill (database):
UPDATE users SET email_address = email WHERE email_address IS NULL;
```

```
Result:
users: [id, name, email="foo@bar.com", email_address="foo@bar.com"]
Both columns have data
```

Contract Phase

```
All code updated to use email_address:
SELECT email_address FROM users ✓
Old column unused

Safe to drop:
ALTER TABLE users DROP COLUMN email;

Final schema:
users: [id, name, email_address]
```

Rollback Safety

At any point during migration:

- **Expand phase:** Can rollback (just added column, not required)
- **Migrate phase:** Can rollback (both columns populated)
- **Contract phase:** Can rollback (code still dual-writing)

4. Correct Design & Algorithm

Migration Framework Architecture

```
1. Migration files (versioned):
   migrations/
     001_add_users_table.sql
     002_add_email_column.sql
     003_add_email_address_column.sql

2. Migration tracking table:
   schema_migrations: [version, applied_at]

3. Migration tool:
   - Read migrations folder
   - Check which migrations applied
   - Run pending migrations in order
   - Record completion in schema_migrations

4. Safety checks:
   - Idempotent (can run twice safely)
   - Transactional (rollback on error)
   - Ordered (dependencies respected)
```

Expand-Migrate-Contract Timeline

```
Week 1 (Expand):
- Deploy migration: Add email_address column
- Deploy code: Write to both email and email_address
- Monitor: Ensure no errors

Week 2 (Migrate):
- Run backfill: Copy email → email_address (batched, off-peak)
- Verify: All rows have email_address populated
- Monitor: Check data consistency

Week 3 (Migrate continued):
- Deploy code: Read from email_address, write to both
- Monitor: Application works with new column

Week 4 (Contract preparation):
- Deploy code: Read from email_address, write only to email_address
- Wait: Ensure no rollback needed
- Monitor: Old column unused

Week 5 (Contract):
- Deploy migration: Drop email column
- Final schema: Only email_address exists
- Cleanup complete
```

5. Full Production-Grade Implementation

```typescript
import { Pool, PoolClient } from 'pg';
import fs from 'fs/promises';
import path from 'path';

/**
 * Migration file metadata
 */
interface Migration {
  version: number;
  name: string;
  filename: string;
  sql: string;
}

/**
 * Database migration manager
 */
class MigrationManager {
  constructor(private pool: Pool) {}

  /**
   * Run pending migrations
   */
  async migrate(): Promise<void> {
```

```
    const client = await this.pool.connect();

    try {
      await this.ensureMigrationTable(client);

      const appliedVersions = await this.getAppliedMigrations(client);
      const allMigrations = await this.loadMigrations();
      const pendingMigrations = allMigrations.filter(
        m => !appliedVersions.includes(m.version)
      );

      if (pendingMigrations.length === 0) {
        console.log('[Migration] No pending migrations');
        return;
      }

      console.log(`[Migration] Found ${pendingMigrations.length} pending
migrations`);

      for (const migration of pendingMigrations) {
        await this.runMigration(client, migration);
      }

      console.log('[Migration] All migrations completed successfully');

    } finally {
      client.release();
    }
  }

  /**
   * Create schema_migrations table if not exists
   */
  private async ensureMigrationTable(client: PoolClient): Promise<void> {
    await client.query(`
      CREATE TABLE IF NOT EXISTS schema_migrations (
        version INTEGER PRIMARY KEY,
        name VARCHAR(255) NOT NULL,
        applied_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )
    `);
  }

  /**
   * Get list of already-applied migration versions
   */
  private async getAppliedMigrations(client: PoolClient): Promise<number[]> {
    const result = await client.query('SELECT version FROM schema_migrations ORDER
BY version');
    return result.rows.map(row => row.version);
  }
```

```
  /**
   * Load migration files from disk
   */
  private async loadMigrations(): Promise<Migration[]> {
    const migrationsDir = path.join(__dirname, 'migrations');
    const files = await fs.readdir(migrationsDir);

    const migrations: Migration[] = [];

    for (const filename of files) {
      if (!filename.endsWith('.sql')) continue;

      const match = filename.match(/^(\d+)_(.+)\.sql$/);
      if (!match) {
        console.warn(`[Migration] Skipping invalid filename: ${filename}`);
        continue;
      }

      const version = parseInt(match[1], 10);
      const name = match[2];
      const filepath = path.join(migrationsDir, filename);
      const sql = await fs.readFile(filepath, 'utf-8');

      migrations.push({ version, name, filename, sql });
    }

    // Sort by version
    migrations.sort((a, b) => a.version - b.version);

    return migrations;
  }

  /**
   * Run a single migration
   */
  private async runMigration(client: PoolClient, migration: Migration):
Promise<void> {
    console.log(`[Migration] Running ${migration.filename}...`);

    try {
      await client.query('BEGIN');

      // Execute migration SQL
      await client.query(migration.sql);

      // Record migration as applied
      await client.query(
        'INSERT INTO schema_migrations (version, name) VALUES ($1, $2)',
        [migration.version, migration.name]
      );

      await client.query('COMMIT');
```

```
      console.log(`[Migration] ✓ ${migration.filename} completed`);

    } catch (error) {
      await client.query('ROLLBACK');
      console.error(`[Migration] ✗ ${migration.filename} failed:`, error);
      throw new Error(`Migration ${migration.version} failed: ${error}`);
    }
  }

  /**
   * Rollback last migration (use with caution)
   */
  async rollback(): Promise<void> {
    const client = await this.pool.connect();

    try {
      const lastMigration = await client.query(
        'SELECT version, name FROM schema_migrations ORDER BY version DESC LIMIT 1'
      );

      if (lastMigration.rows.length === 0) {
        console.log('[Migration] No migrations to rollback');
        return;
      }

      const { version, name } = lastMigration.rows[0];
      console.warn(`[Migration] Rolling back ${version}_${name}`);

      // Load rollback SQL (if exists)
      const rollbackFile = path.join(__dirname, 'migrations',
`${version}_${name}_down.sql`);

      try {
        const rollbackSQL = await fs.readFile(rollbackFile, 'utf-8');

        await client.query('BEGIN');
        await client.query(rollbackSQL);
        await client.query('DELETE FROM schema_migrations WHERE version = $1',
[version]);
        await client.query('COMMIT');

        console.log(`[Migration] ✓ Rolled back ${version}_${name}`);

      } catch (error: any) {
        if (error.code === 'ENOENT') {
          console.error(`[Migration] Rollback file not found: ${rollbackFile}`);
        } else {
          await client.query('ROLLBACK');
          throw error;
        }
      }
```

```
    } finally {
      client.release();
    }
  }
}

/**
 * Batched backfill for large tables
 */
class BackfillManager {
  constructor(private pool: Pool) {}

  /**
   * Backfill data in batches to avoid locks
   */
  async backfill(
    tableName: string,
    sourceColumn: string,
    targetColumn: string,
    batchSize: number = 1000
  ): Promise<void> {
    console.log(`[Backfill] Starting ${tableName}.${sourceColumn} →
${targetColumn}`);

    let processedRows = 0;
    let hasMore = true;

    while (hasMore) {
      const result = await this.pool.query(`
        UPDATE ${tableName}
        SET ${targetColumn} = ${sourceColumn}
        WHERE id IN (
          SELECT id FROM ${tableName}
          WHERE ${targetColumn} IS NULL
          LIMIT $1
        )
      `, [batchSize]);

      processedRows += result.rowCount || 0;
      hasMore = (result.rowCount || 0) > 0;

      console.log(`[Backfill] Processed ${processedRows} rows...`);

      // Pause between batches to avoid overwhelming database
      if (hasMore) {
        await new Promise(resolve => setTimeout(resolve, 100));
      }
    }

    console.log(`[Backfill] ✓ Completed. Total rows: ${processedRows}`);
  }
```

```
}

// Initialize
const pool = new Pool({
  host: 'localhost',
  database: 'myapp',
  user: 'postgres',
  password: 'password',
  max: 20,
});

const migrationManager = new MigrationManager(pool);
const backfillManager = new BackfillManager(pool);

// CLI commands
if (require.main === module) {
  const command = process.argv[2];

  (async () => {
    try {
      if (command === 'migrate') {
        await migrationManager.migrate();
      } else if (command === 'rollback') {
        await migrationManager.rollback();
      } else if (command === 'backfill') {
        const [table, source, target] = process.argv.slice(3);
        await backfillManager.backfill(table, source, target);
      } else {
        console.log('Usage:');
        console.log('  npm run migrate:up      - Run pending migrations');
        console.log('  npm run migrate:down    - Rollback last migration');
        console.log('  npm run backfill <table> <source> <target> - Backfill data');
      }

      process.exit(0);
    } catch (error) {
      console.error('Migration failed:', error);
      process.exit(1);
    }
  })();
}

/**
 * Example migration files
 */

// migrations/001_create_users_table.sql
const migration001 = `
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE,
```

```
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_users_email ON users(email);
`;

// migrations/002_add_email_address_column.sql (Expand)
const migration002 = `
-- Add new column (nullable, backward compatible)
ALTER TABLE users ADD COLUMN email_address VARCHAR(255);

-- Index for new column
CREATE INDEX idx_users_email_address ON users(email_address);
`;

// migrations/003_make_email_address_not_null.sql (After backfill)
const migration003 = `
-- Verify all rows have email_address
DO $$
BEGIN
  IF EXISTS (SELECT 1 FROM users WHERE email_address IS NULL) THEN
    RAISE EXCEPTION 'Cannot make email_address NOT NULL: NULL values exist';
  END IF;
END $$;

-- Make NOT NULL
ALTER TABLE users ALTER COLUMN email_address SET NOT NULL;

-- Add unique constraint
ALTER TABLE users ADD CONSTRAINT users_email_address_unique UNIQUE (email_address);
`;

// migrations/004_drop_old_email_column.sql (Contract)
const migration004 = `
-- Drop old column (after all code updated)
ALTER TABLE users DROP COLUMN email;

-- Drop old index
DROP INDEX IF EXISTS idx_users_email;
`;

/**
 * Example: Dual-write application code (Migrate phase)
 */
class UserRepository {
  constructor(private pool: Pool) {}

  async updateEmail(userId: number, newEmail: string): Promise<void> {
    // Dual-write: Update both old and new columns
    await this.pool.query(
      `UPDATE users
       SET email = $1, email_address = $1
```

```
        WHERE id = $2`,
       [newEmail, userId]
     );
   }

   async getUser(userId: number): Promise<any> {
     const result = await this.pool.query(
       // Read from new column, fallback to old
       `SELECT id, name,
               COALESCE(email_address, email) as email
        FROM users
        WHERE id = $1`,
       [userId]
     );

     return result.rows[0];
   }
}
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Adding a Column

```
-- Expand
ALTER TABLE users ADD COLUMN phone VARCHAR(20);

-- Migrate (application code dual-writes)
-- Backfill existing rows
UPDATE users SET phone = '' WHERE phone IS NULL;

-- Contract
ALTER TABLE users ALTER COLUMN phone SET NOT NULL;
```

Pattern 2: Renaming a Column

```
-- Expand: Add new column
ALTER TABLE users ADD COLUMN full_name VARCHAR(255);

-- Migrate: Copy data
UPDATE users SET full_name = name;

-- Contract: Drop old column
ALTER TABLE users DROP COLUMN name;
```

Pattern 3: Changing Column Type

```
-- Expand: Add new column with new type
ALTER TABLE products ADD COLUMN price_cents INTEGER;
```

```sql
-- Migrate: Convert data
UPDATE products SET price_cents = (price * 100)::INTEGER;

-- Contract: Drop old column
ALTER TABLE products DROP COLUMN price;
```

7. Failure Modes & Edge Cases

Large Tables

Problem: Backfill locks table, blocks queries.

Solution: Batch updates (1000 rows at a time), run off-peak.

Foreign Keys

Problem: Can't drop column referenced by foreign key.

Solution: Drop foreign key first, then column.

Deployed Code Mismatch

Problem: Old code expects old schema after migration.

Solution: Always keep old columns during code deployment window.

8. Performance Characteristics & Tradeoffs

Migration Speed

- **Add column:** Instant (no data rewrite)
- **Add NOT NULL column:** Slow (table rewrite in PostgreSQL <11)
- **Backfill 10M rows:** 5-30 minutes (depends on batch size)

Downtime

- **Expand-migrate-contract:** Zero downtime
- **Direct changes:** Requires downtime

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: Breaking Change in Single Migration

Fix: Use expand-migrate-contract pattern.

Mistake 2: Backfill During Peak Traffic

Fix: Run backfills off-peak, in batches.

Mistake 3: No Rollback Plan

Fix: Keep old columns until all code updated.

Mistake 4: Manual Migration Tracking

Fix: Use schema_migrations table and automated tool.

Mistake 5: No Migration Testing

Fix: Test migrations on staging environment first.

10. When NOT to Use This (Anti-Patterns)

New Databases

Empty databases don't need expand-migrate-contract, can apply schema directly.

Read-Only Tables

Historical data tables can have breaking changes (no active code).

Development Environment

Localhost can drop/recreate database freely.

11. Related Concepts (With Contrast)

ORM Migrations

Difference: ORM (Sequelize, TypeORM) generates migrations from code, manual migrations are SQL files.

Blue-Green Deployment

Related: Schema must support both blue and green versions simultaneously.

Feature Flags

Related: Can hide new columns behind flags during migration.

12. Production Readiness Checklist

Migration Infrastructure

- [ ] Migrations versioned and ordered
- [ ] schema_migrations table tracks applied migrations
- [ ] Automated migration tool
- [ ] Migrations are idempotent

Safety

- [ ] Migrations tested on staging
- [ ] Expand-migrate-contract pattern for breaking changes
- [ ] Rollback plan for each migration
- [ ] Backfills run in batches (off-peak)

Monitoring

- [ ] Migration failures alert on-call
- [ ] Track migration duration

- [ ] Monitor database locks during migration
- [ ] Verify data consistency after backfill

Process

- [ ] Migrations reviewed in code review
- [ ] Deployment runbook includes migration steps
- [ ] Communication plan for long-running migrations
- [ ] Rollback procedure documented and tested