MongoDB to PostgreSQL Migration Plan

Phase 1: Analysis and Schema Design

This is the most critical phase. A thorough analysis and a well-designed relational schema will prevent significant issues later.

1. **Analyze MongoDB Schema:**
   - Document all MongoDB collections and the structure of the documents within them.
   - Identify embedded documents and arrays. These will need to be modeled as separate tables or using specific PostgreSQL types (like `JSONB` or `ARRAY`).
   - Map out relationships between collections (e.g., manual references using `ObjectId`).
2. **Design Target PostgreSQL Schema:**
   - For each MongoDB collection, design one or more corresponding PostgreSQL tables.
   - Define columns with appropriate PostgreSQL data types. Create a mapping table.

MongoDB Type	PostgreSQL Type	Notes
`ObjectId`	`UUID` or `BIGSERIAL`	`UUID` is a good choice to maintain global uniqueness. `BIGSERIAL` is simpler for auto-incrementing primary keys.
`String`	`TEXT` or `VARCHAR(n)`	Use `TEXT` unless you have a strict length requirement.
`Number`	`INTEGER`, `BIGINT`, `NUMERIC`	Choose based on the range and precision of the numbers.
`Date`	`TIMESTAMP WITH TIME ZONE`	`TIMESTAMPTZ` is best practice for storing time data.
`Boolean`	`BOOLEAN`	Direct mapping.
`Array`	`ARRAY` type (e.g., `TEXT[]`) or a separate join table.	A join table is more flexible and normalized for complex objects.
`Object` / Embedded	`JSONB` or a separate table with a foreign key	`JSONB` is powerful for unstructured data, but a separate table is better for

MongoDB Type	PostgreSQL Type	Notes
Doc	relationship.	data you need to query or index directly.

3. **Define Relationships:**
    - Convert MongoDB document references into PostgreSQL foreign key constraints to enforce relational integrity.
    - Use join tables to model many-to-many relationships.
4. **Choose Migration Tools:**
    - Decide between using a dedicated ETL tool (e.g., Pentaho, Talend) or writing custom migration scripts (e.g., in Python with `pymongo` and `psycopg2`, or Node.js with `mongodb` and `pg`). Custom scripts offer more control, which is often necessary for complex transformations.

---

§ Phase 2: Environment Setup and Schema Implementation

1. **Set up PostgreSQL:** Install and configure a PostgreSQL server for development and testing. Create a new database, users, and roles with appropriate permissions.
2. **Write DDL Scripts:** Based on the design from Phase 1, write the SQL Data Definition Language (DDL) scripts.
3. **Execute DDL:** Run the scripts to create all tables, indexes, sequences, views, and foreign key constraints in the target PostgreSQL database.

```
-- Example DDL for migrating a 'users' collection

CREATE EXTENSION IF NOT EXISTS "uuid-ossp"; -- To generate UUIDs

CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    email TEXT NOT NULL UNIQUE,
    password_hash TEXT NOT NULL,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

-- Example for a related 'posts' collection
CREATE TABLE posts (
```

```
    id BIGSERIAL PRIMARY KEY,
    author_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    title TEXT NOT NULL,
    content TEXT,
    published_at TIMESTAMPTZ,
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE INDEX idx_posts_author_id ON posts(author_id);
```

Phase 3: Data Migration (ETL Process)

This phase involves writing and running scripts to Extract, Transform, and Load data.

1. **Extract:** Write code to connect to the MongoDB database and read all documents from a source collection.
2. **Transform:** For each MongoDB document, write transformation logic to:
   - Flatten the document structure to fit the relational model.
   - Convert data types (e.g., `ObjectId` to a string representation for `UUID` or mapping to a new `SERIAL` key).
   - Handle nested arrays and objects by preparing data for insertion into related tables.
3. **Load:** Write code to connect to PostgreSQL and insert the transformed data into the appropriate tables.
   - Be mindful of the insertion order to respect foreign key constraints (e.g., insert `users` before their `posts`).
   - Use transactions to ensure data is inserted atomically.

Phase 4: Application Code Refactoring

This is a significant development effort that can be done in parallel with data migration scripting.

1. **Update Dependencies:** Replace the MongoDB driver and ODM/ORM (e.g., Mongoose) with a PostgreSQL driver and ORM (e.g., Sequelize, TypeORM, pg-promise).

2. **Rewrite Data Access Layer:** Refactor all database queries from the MongoDB query syntax to SQL. This includes simple CRUD operations and complex aggregation pipelines.
    - **Before (Mongoose):** `User.find({ age: { $gt: 21 } })`
    - **After (Sequelize):** `User.findAll({ where: { age: { [Op.gt]: 21 } } })`
    - **After (Raw SQL):** `SELECT * FROM users WHERE age > 21`
3. **Adapt Business Logic:** Adjust any application logic that relied on MongoDB-specific features or schemaless flexibility.

Phase 5: Validation and Testing

1. **Data Integrity Verification:**
    - Compare row counts between MongoDB collections and PostgreSQL tables.
    - Write scripts to perform spot checks, comparing a sample of records from the source and target to ensure data was not corrupted.
    - Validate that all relations were created correctly.
2. **Application Testing:**
    - **Unit Tests:** Update and run all unit tests for the refactored data access layer.
    - **Integration Tests:** Perform end-to-end testing of all application features to ensure they work correctly with PostgreSQL.
    - **Performance Testing:** Benchmark the application against the new database. Identify and optimize slow-running SQL queries using tools like `EXPLAIN ANALYZE`. Add indexes where necessary.

Phase 6: Deployment and Cutover

1. **Plan the Cutover:**
    - **Strategy:** Decide on a "big bang" (downtime required) or a "live" migration strategy. For most applications, a scheduled downtime is safer and simpler.
    - **Timeline:** Schedule the migration during a low-traffic period.
2. **Execute the Final Migration:**
    - Put the application into maintenance mode.
    - Perform a final, complete data migration from the production MongoDB to the production PostgreSQL database to ensure data is up-to-date.

- Run verification scripts to confirm the integrity of the production data.

3. **Go Live:**
    - Update the application's environment variables to point to the new PostgreSQL database.
    - Deploy the refactored application code.
    - Disable maintenance mode and bring the application back online.
4. **Post-Migration:**
    - Closely monitor application logs and performance metrics for any issues.
    - Keep the MongoDB database online in a read-only state for a short period as a fallback, but do not allow the application to write to it.
    - Once confident, decommission the old MongoDB database.