ADVANCED DATABASE SYSTEMS - FINAL EXAM STUDY GUIDE

Topics 08-13 Comprehensive Review

This study guide covers the advanced database topics starting from PostgreSQL through MongoDB and geospatial database operations. Use this to prepare for your final exam.

TOPIC 08: PostgreSQL Setup and Security

KEY CONCEPTS:

1. PostgreSQL vs SQLite

- **PostgreSQL**: Client-server, multi-user, production-grade RDBMS
- **SQLite**: File-based, single-writer, embedded database
- PostgreSQL supports concurrent users with MVCC (Multi-Version Concurrency Control)
- PostgreSQL has robust security (role-based access control, SSL/TLS)

2. Installation & Setup

- **macOS**: `brew install postgresql@15`
- **Linux**: `apt install postgresql postgresql-contrib`
- **Start service**: `brew services start postgresql@15` (macOS)
- **Default port**: 5432

3. User Management

- Default superuser: `postgres`
- Create users: `CREATE USER username WITH PASSWORD 'password';`
- Grant privileges: `GRANT SELECT, INSERT, UPDATE, DELETE ON table TO user;`
- Create dedicated application users for applications

4. Database Schema Differences from SQLite

- `SERIAL` instead of `AUTOINCREMENT` for auto-increment primary keys
- `VARCHAR(n)` instead of `TEXT` for variable-length strings
- Parameter placeholders: `%s` (PostgreSQL) vs `?` (SQLite)
- Explicit `COMMIT` required (transactions default to autocommit=False)
- `CHECK` constraints enforced at database level
- `UNIQUE` constraints prevent duplicates

5. Python Connection (psycopg2)

- Install: `pip install psycopg2-binary`

- Connection parameters: host, database, user, password
- `RealDictCursor` returns rows as dictionaries
- Must explicitly commit: `connection.commit()`
- Use try/except/finally for error handling
- Rollback on error: `connection.rollback()`

6. SQL Injection Prevention

- Always use parameter placeholders: `?` (SQLite) or `%s` (PostgreSQL)
- NEVER use string concatenation or f-strings for queries
- Placeholders separate SQL code from data values
- Database driver properly escapes values

SAFE (parameterized):

```
cursor.execute("SELECT * FROM pet WHERE owner = ?", (owner,))
cursor.execute("SELECT * FROM pet WHERE owner = %s", (owner,))
```

DANGEROUS (string concatenation):

```
cursor.execute(f"SELECT * FROM pet WHERE owner = '{owner}'")  # DON'T DO
THIS!
cursor.execute("SELECT * FROM pet WHERE owner = '" + owner + "'")  #
DON'T!
```

Attack example: If `owner = "Greg' OR '1'='1"` with string concatenation, the query becomes:

```
SELECT * FROM pet WHERE owner = 'Greg' OR '1'='1'
```

This returns ALL pets because `'1'='1'` is always true!

With parameterized queries, the malicious input is treated as a literal string value, not executable SQL code.

7. Foreign Key Constraints

- `ON DELETE RESTRICT`: Prevents deletion if referenced
- `ON DELETE CASCADE`: Deletes related records
- `ON UPDATE CASCADE`: Updates related records
- Better referential integrity than SQLite

8. Common PostgreSQL Commands

- `\l` - List all databases
- `\c database_name` - Connect to database
- `\dt` - List tables

- `\d table_name` - Describe table structure
- `\du` - List users/roles
- `SELECT current_user;` - Show current user
- `SELECT current_database();` - Show current database

9. Backup and Restore

- **Backup**: `pg_dump -U postgres database_name > backup.sql`
- **Backup compressed**: `pg_dump -U postgres -Fc database_name > backup.dump`
- **Restore**: `psql -U postgres database_name < backup.sql`
- **Restore compressed**: `pg_restore -U postgres -d database_name backup.dump`

EXAM TIPS:

- Know the difference between SQLite and PostgreSQL use cases
- Be able to write SQL to create users and grant permissions
- Know how to convert SQLite code to PostgreSQL (SERIAL, %s, VARCHAR)
- Understand transaction handling and explicit commits
- Always use parameterized queries to prevent SQL injection
- Know foreign key constraint options (RESTRICT, CASCADE)

SAMPLE QUESTIONS:

Q: How do you grant a user permission to select and insert on all tables?
A: `GRANT SELECT, INSERT ON ALL TABLES IN SCHEMA public TO username;`
`GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO username;`

Q: What is the PostgreSQL equivalent of SQLite's AUTOINCREMENT?
A: `SERIAL` (or `SERIAL PRIMARY KEY`)

Q: What parameter placeholder does PostgreSQL use instead of SQLite's '?'?
A: PostgreSQL uses `%s` for parameter placeholders in queries.

Q: What is SQL injection and how do parameter placeholders prevent it?
A: SQL injection is when malicious SQL code is inserted through user input. Parameter placeholders (`?` or `%s`) prevent it by separating SQL code from data, treating all input as literal values rather than executable code. Never use string concatenation to build queries!

TOPIC 09: Mongita Introduction - MongoDB Concepts

KEY CONCEPTS:

1. What is Mongita?

- Lightweight, pure-Python implementation of MongoDB API
- No server required - runs in Python process
- Uses MongoDB syntax - code works identically in real MongoDB
- Stores data locally in files
- Perfect for learning and prototyping

2. Document-Oriented Paradigm Shift

- Documents instead of rows
- Collections instead of tables
- Fields instead of columns
- Flexible schema - documents can have different fields
- No pre-defined schema required
- JSON-like structure (BSON internally)

3. Basic Operations (CRUD)

CREATE:

- `insert_one({'field': 'value'})` - Insert single document
- `insert_many([{...}, {...}])` - Insert multiple documents
- Documents auto-created with `_id` field (ObjectId)

READ:

- `find()` - Returns cursor to all documents
- `find({'field': 'value'})` - Query with conditions
- `find_one({'field': 'value'})` - Returns single document
- `count_documents({})` - Count all documents
- `{}` (empty dict) matches all documents

UPDATE:

- `update_one(query, {'$set': {'field': 'new_value'}})`
- `$set` operator adds or updates fields
- Other operators: `$unset`, `$inc`, `$push`, `$pull`

DELETE:

- `delete_one(query)` - Removes first matching document
- `delete_many(query)` - Removes all matching documents

4. Query Operators

- `$gt`: Greater than
- `$lt`: Less than
- `$gte`: Greater than or equal
- `$lte`: Less than or equal
- `$ne`: Not equal
- `$in`: Value in array

Example: `{'age': {'$gt': 5}}` finds documents where age > 5

5. Cursors

- `find()` returns a cursor (iterator), not results directly

- Must convert to list: `list(collection.find())`
- Cursors are consumed when iterated - they can only be read once
- Cannot iterate a cursor twice - must re-query or store results

Example of cursor consumption:

```
cursor = collection.find({'age': {'$gt': 5}})
list(cursor)  # [results returned]
list(cursor)  # [] - Empty! Cursor already consumed
```

Solutions:

```
# Option 1: Store results in a list initially
results = list(collection.find({'age': {'$gt': 5}}))
# Now you can iterate over 'results' multiple times

# Option 2: Re-query each time you need the data
cursor = collection.find({'age': {'$gt': 5}})  # Fresh cursor
list(cursor)  # Results returned again
```

Key insight: Cursors are like one-time-use iterators. Once exhausted, they're empty and cannot be reset.

6. ObjectId

- Unique identifier automatically generated for each document
- Stored in `_id` field
- Like auto-increment primary key in SQL
- 12-byte identifier, includes timestamp
- Can convert: `str(object_id)` and `ObjectId(string)`

7. Schema Flexibility

- Documents in same collection can have different fields
- No schema enforcement
- Schema evolves naturally with application
- Add fields to document without ALTER TABLE
- Example: Two documents with completely different structures are valid

8. Auto-creation

- Databases created on first access
- Collections created on first use
- No CREATE DATABASE or CREATE TABLE commands needed

9. Comparison with SQL

SQL	MongoDB
`INSERT INTO table VALUES (...)`	`collection.insert_one({...})`
`SELECT * FROM table`	`collection.find()`
`SELECT * WHERE col > 1`	`collection.find({'field': {'$gt': 1}})`
`UPDATE table SET col=val WHERE id=1`	`collection.update_one({'id': 1}, {'$set': {'col': val}})`
`DELETE FROM table WHERE id=1`	`collection.delete_one({'id': 1})`
`SELECT COUNT(*) FROM table`	`collection.count_documents({})`

EXAM TIPS:

- Understand document vs. relational paradigm
- Know CRUD operations and syntax
- Remember query operators ($gt, $set, etc.)
- Understand cursor behavior and consumption
- Know ObjectId purpose and usage
- Recognize schema flexibility advantages/disadvantages
- Be able to convert SQL operations to MongoDB

SAMPLE QUESTIONS:

Q: What happens if you try to iterate a cursor twice?
A: The second iteration returns empty results because cursors are consumed after the first iteration. You must re-query to get fresh results.

Q: How do you add a new field to an existing document?
A: `collection.update_one({'_id': doc_id}, {'$set': {'new_field': 'value'}})`
No schema modification needed - field is added dynamically.

Q: What is the MongoDB equivalent of SELECT * FROM table WHERE age > 18?
A: `list(collection.find({'age': {'$gt': 18}}))`

TOPIC 10: MongoDB with Flask Applications

KEY CONCEPTS:

1. Application Architecture

- Flask web application with Mongita backend
- database.py: All database operations
- app.py: Flask routes and HTTP handling
- create-database.py: Initialize with sample data
- Separation of concerns: database logic separate from web logic

2. Document References (Manual Joins)

- MongoDB has no built-in JOIN operations
- Use manual references: store ObjectId of one document in another
- Example: pet document stores kind_id referencing kind document
- Must explicitly fetch and combine related data in code

Pattern:

```
pet = {'name': 'Suzy', 'kind_id': ObjectId('...')}
kind = kind_collection.find_one({'_id': pet['kind_id']})
pet['kind_name'] = kind['kind_name']
```

3. ObjectId String Conversion

- MongoDB stores ObjectId objects
- Web forms/URLs use string IDs
- Convert string to ObjectId for queries: `ObjectId(id_string)`
- Convert ObjectId to string for display: `str(object_id)`
- Always convert before querying: `find_one({'_id': ObjectId(id)})`

4. CRUD Operations in Application Context

CREATE:

```
def create_pet(data):
    data['kind_id'] = ObjectId(data['kind_id'])
    pets_collection.insert_one(data)
```

READ:

```
def retrieve_pets():
    pets = list(pets_collection.find())
    # Manual join to get kind information
    for pet in pets:
        kind = kind_collection.find_one({'_id': pet['kind_id']})
        pet['kind_name'] = kind['kind_name']
    return pets
```

UPDATE:

```
def update_pet(id, data):
    pets_collection.update_one(
        {'_id': ObjectId(id)},
```

```
        {'$set': data}
    )
```

DELETE:

```
def delete_pet(id):
    pets_collection.delete_one({'_id': ObjectId(id)})
```

5. Data Integrity Considerations

- MongoDB doesn't enforce referential integrity automatically
- No foreign key constraints
- Application must handle integrity
- Example: Check if pets reference a kind before deleting kind
- Manual validation required

6. Flask Integration

- RESTful routing patterns
- `GET /list` - Display all items
- `GET /create` - Show form
- `POST /create` - Process form submission
- `GET /update/<id>` - Show update form
- `POST /update/<id>` - Process update
- `GET /delete/<id>` - Delete item

7. Form Handling

- `request.form` to get form data
- Convert to dict: `data = dict(request.form)`
- Type conversion: `data['age'] = int(data['age'])`
- Redirect after POST: `return redirect(url_for('get_list'))`

8. Testing Patterns

- Each CRUD operation has a test function
- Tests verify data types and structure
- Tests verify actual values
- Tests clean up after themselves (create then delete)
- Run tests: `python database.py`

9. Transition to Real MongoDB

- Change import: `from pymongo import MongoClient`
- Change connection: `client = MongoClient('mongodb://localhost:27017/')`
- All other code remains the same

- Demonstrates Mongita's value as learning tool

10. Collections Used

- `pets_collection`: Stores pet records
- `kind_collection`: Stores pet kind information
- Two-collection model similar to two-table SQL design

EXAM TIPS:

- Understand manual joins and why they're necessary
- Know ObjectId string conversion pattern
- Understand application-level referential integrity
- Know how to implement CRUD operations with Flask
- Recognize the difference between database operations and web routes
- Understand testing patterns
- Know how Mongita code translates to MongoDB

SAMPLE QUESTIONS:

Q: How do you perform a "join" in MongoDB to get pet and kind information?
A: Manually: fetch pet document, use `pet['kind_id']` to query kind_collection, then merge the data into the pet dictionary.

Q: Why must you convert string IDs to ObjectId before querying?
A: MongoDB stores `_id` as ObjectId type. String comparison won't match ObjectId. Must convert: `pets_collection.find_one({'_id': ObjectId(id_string)})`

Q: How does MongoDB handle referential integrity?
A: It doesn't automatically. The application must check for references before deleting. No built-in foreign key constraints like SQL databases.

TOPIC 11: MongoDB Atlas (Cloud-Hosted MongoDB)

KEY CONCEPTS:

1. MongoDB Atlas Overview

- Cloud-hosted MongoDB Database-as-a-Service (DBaaS)
- Managed by MongoDB Inc.
- Free tier available (M0 cluster)
- No local installation or server management required
- Automatic backups, monitoring, and scaling

2. Connection to Atlas

- Connection string format: `mongodb+srv://username:password@cluster.mongodb.net/?appName=AppName`
- Use pymongo instead of Mongita

- Connection: `client = pymongo.MongoClient(connection_string)`
- Handle connection errors with try/except

3. pymongo vs Mongita

- Same API, different implementation
- pymongo connects to real MongoDB servers
- Mongita is local, in-memory/disk-based
- Code is interchangeable

```
# Mongita:
client = MongitaClientDisk()

# pymongo:
client = pymongo.MongoClient(uri)
```

4. Connection Best Practices

- Store connection strings in environment variables or config files
- Add .env files to .gitignore if used
- Use database users with appropriate privileges

5. Atlas Features

- Automatic backups
- Performance monitoring
- Cloud provider choice (AWS, Azure, Google Cloud)
- Horizontal scaling (sharding)
- Replica sets for high availability
- Geographic distribution

6. Error Handling

- `ConfigurationError`: Invalid URI or connection string
- `ConnectionFailure`: Cannot reach server
- `OperationFailure`: Authentication or permission issues
- Always wrap connection in try/except

7. Same Application Code

- Flask app.py remains mostly unchanged
- Only connection initialization changes
- database.py CRUD operations identical
- Demonstrates cloud migration simplicity

EXAM TIPS:

- Understand Atlas as cloud-hosted MongoDB
- Know connection string format
- Store credentials appropriately (not hardcoded)
- Understand error handling for cloud connections
- Know the difference between Mongita and pymongo usage
- Recognize advantages of cloud-hosted databases

SAMPLE QUESTIONS:

Q: What is MongoDB Atlas?
A: Cloud-hosted MongoDB Database-as-a-Service (DBaaS) managed by MongoDB Inc., providing automatic backups, monitoring, and scaling without server management.

Q: What's the main code change when moving from Mongita to MongoDB Atlas?
A: Change the client initialization:
From: `client = MongitaClientDisk()`
To: `client = pymongo.MongoClient('mongodb+srv://...')`

Q: Where should you store connection strings with passwords?
A: Store them in environment variables or config files (not in version control). Add .env files to .gitignore to prevent accidental commits.

TOPIC 12: MongoDB Aggregation Pipeline

KEY CONCEPTS:

1. Aggregation Framework

- Pipeline of data transformations
- Each stage processes and passes results to next stage
- More powerful than simple queries
- Used for complex data analysis and transformations
- Similar to SQL GROUP BY, JOIN, and transformations

2. Aggregation Pipeline Stages

$lookup - Perform joins between collections

- Equivalent to SQL JOIN
- Joins documents from two collections
- Syntax:

```
{
  "$lookup": {
    "from": "other_collection",
    "localField": "field_in_current",
    "foreignField": "field_in_other",
    "as": "output_array_field"
```

```
    }
}
```

- Result is array of matched documents

$unwind - Deconstruct array field

- Converts array field into multiple documents
- Each array element becomes separate document
- Often used after $lookup to flatten results
- Syntax: `{"$unwind": "$array_field"}`

$project - Select and transform fields

- Like SQL SELECT clause
- Includes/excludes fields: 1 = include, 0 = exclude
- Can create computed fields
- Can rename fields using expressions
- Syntax:

```
{
  "$project": {
    "field1": 1,
    "new_name": "$old_field",
    "computed": {"$add": ["$field1", "$field2"]}
  }
}
```

$match - Filter documents

- Like SQL WHERE clause
- Filters documents in pipeline
- Best practice: Use early in pipeline for performance

$group - Group and aggregate

- Like SQL GROUP BY
- Aggregation operators: $sum, $avg, $min, $max, $count

$sort - Order results

- Like SQL ORDER BY
- 1 = ascending, -1 = descending

3. Example Aggregation: Pets with Kind Information

```
pets_collection.aggregate([
  {
```

```
    "$lookup": {
      "from": "kind_collection",
      "localField": "kind_id",
      "foreignField": "_id",
      "as": "kind_info"
    }
  },
  {
    "$unwind": "$kind_info"
  },
  {
    "$project": {
      "name": 1,
      "age": 1,
      "owner": 1,
      "kind_name": "$kind_info.kind_name",
      "food": "$kind_info.food",
      "noise": "$kind_info.noise"
    }
  }
])
```

4. Aggregation vs Manual Joins

- Manual join: Loop through documents, fetch related data
- Aggregation $lookup: Database performs join efficiently
- Aggregation is faster for large datasets
- Aggregation pipelines run on database server
- Manual joins transfer more data to application

5. Conditional Logic in Aggregation

- `$cond`: Conditional expression (if-then-else)
- Syntax:

```
{
  "$cond": {
    "if": {"$lt": ["$age", 3]},
    "then": "young",
    "else": "adult"
  }
}
```

- Can create computed fields based on conditions

6. Accessing Nested Fields

- Use dot notation: `"$kind_info.kind_name"`

- Access fields within embedded documents
- Access array elements: `"$array.0"` (first element)

7. Aggregation Result

- Returns cursor (like find())
- Must convert to list: `list(collection.aggregate([...]))`
- Each stage transforms documents
- Pipeline processes sequentially

EXAM TIPS:

- Understand aggregation pipeline concept (stages)
- Know $lookup syntax and purpose (joins)
- Know $unwind purpose (flatten arrays)
- Know $project for field selection and transformation
- Understand pipeline order matters
- Be able to write basic aggregation pipelines
- Recognize when to use aggregation vs manual queries
- Understand dot notation for nested fields

SAMPLE QUESTIONS:

Q: What is the purpose of $lookup in an aggregation pipeline?
A: $lookup performs joins between collections, similar to SQL JOIN. It fetches matching documents from another collection based on field matching.

Q: Why use $unwind after $lookup?
A: $lookup returns an array of matched documents. $unwind deconstructs this array so each matched document becomes a separate document in the pipeline, making it easier to work with the joined data.

Q: How would you filter pets with age > 5 in an aggregation pipeline?
A: Add a $match stage: `{"$match": {"age": {"$gt": 5}}}`

Q: What does this projection do: {"$project": {"name": 1, "kind": "$kind_info.kind_name"}}?
A: Includes the name field and creates a new "kind" field with the value from the nested "kind_name" field inside "kind_info".

TOPIC 13: GIS (Geographic Information System) Operators

KEY CONCEPTS:

1. Geospatial Data

- Represents locations, areas, and boundaries on Earth
- Stored as coordinates (longitude, latitude)
- GeoJSON format for geographic data
- Used for mapping, location services, geofencing

2. GeoJSON Format

- JSON format for encoding geographic data structures
- Types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon

Point:

```
{
  "type": "Point",
  "coordinates": [longitude, latitude]
}
```

Polygon:

```
{
  "type": "Polygon",
  "coordinates": [
    [[lon1, lat1], [lon2, lat2], [lon3, lat3], [lon1, lat1]]
  ]
}
```

Note: GeoJSON uses `[longitude, latitude]` order (not lat, lon!)

3. MongoDB Geospatial Indexes

- Required for geospatial queries
- 2dsphere index for spherical (Earth) geometry
- Create index: `collection.create_index([("location", "2dsphere")])`
- Enable efficient geospatial queries

4. MongoDB Geospatial Operators

$geoIntersects - Check if geometries intersect

- Tests if point is inside polygon (geofencing)
- Syntax:

```
{
  "area": {
    "$geoIntersects": {
      "$geometry": {
        "type": "Point",
        "coordinates": [lon, lat]
      }
    }
  }
}
```

- Use case: "Is this location inside this zone?"

$geoWithin - Find points within area

- Find all points inside a polygon
- Syntax:

```
{
  "location": {
    "$geoWithin": {
      "$geometry": {polygon_definition}
    }
  }
}
```

- Use case: "Find all stores in this delivery zone"

$near - Find points near a location

- Returns points sorted by distance
- Can specify max distance
- Use case: "Find nearest 5 restaurants"

$nearSphere - Like $near but for spherical geometry

- More accurate for Earth coordinates

5. Geofencing

- Virtual perimeter around geographic area
- Detect when point enters/exits zone
- Example: delivery zones, restricted areas, location-based triggers

Implementation:

```
def in_zone(lat, lon):
    match = zones.find_one({
        "area": {
            "$geoIntersects": {
                "$geometry": {
                    "type": "Point",
                    "coordinates": [lon, lat]
                }
            }
        }
    })
    return match["zone"] if match else None
```

6. Zone Analysis

- Group points by geographic zone
- Determine which zone contains each point
- Example: Assign customers to service territories

Pattern:

```
for place in places.find():
    for zone in zones:
        if point_in_polygon(place.location, zone.boundary):
            assign_to_zone(place, zone)
```

7. PostgreSQL PostGIS Extension

- Add geospatial capabilities to PostgreSQL
- Enable: `CREATE EXTENSION IF NOT EXISTS postgis;`
- Geometry data type: `GEOMETRY(Point, 4326)`
- SRID 4326 = WGS 84 (standard GPS coordinates)

8. PostGIS Functions

- `ST_Point(lon, lat)` - Create point
- `ST_SetSRID(geom, srid)` - Set spatial reference
- `ST_DWithin(geom1, geom2, distance)` - Check if within distance
- `ST_Distance(geom1, geom2)` - Calculate distance
- `ST_Intersects(geom1, geom2)` - Check if intersects

Example:

```
SELECT name FROM locations
WHERE ST_DWithin(
  geom::geography,
  ST_SetSRID(ST_Point(-73.935200, 40.730700), 4326)::geography,
  1000  -- meters
);
```

9. Geography vs Geometry

- **geometry**: Planar (flat) coordinates, faster
- **geography**: Spherical (Earth surface), more accurate for distance
- Cast with `::geography` or `::geometry`

10. Common Use Cases

- Store locator: Find nearest locations

- Delivery zones: Check if address in service area
- Ride sharing: Match drivers to riders by proximity
- Real estate: Search properties within area
- Fleet tracking: Monitor vehicle locations
- Weather: Associate observations with regions

EXAM TIPS:

- Remember GeoJSON coordinate order: [longitude, latitude]
- Know difference between $geoIntersects and $geoWithin
- Understand geofencing concept and implementation
- Know when to use 2dsphere index
- Understand PostGIS vs MongoDB geospatial capabilities
- Know common PostGIS functions (ST_Point, ST_DWithin)
- Understand geography vs geometry in PostGIS
- Be able to write basic geofencing queries

SAMPLE QUESTIONS:

Q: What is the coordinate order in GeoJSON?
A: `[longitude, latitude]` - longitude first, then latitude

Q: How do you check if a point is inside a polygon in MongoDB?
A: Use $geoIntersects operator:

```
collection.find_one({
  "area": {
    "$geoIntersects": {
      "$geometry": {"type": "Point", "coordinates": [lon, lat]}
    }
  }
})
```

Q: What index is required for MongoDB geospatial queries?
A: 2dsphere index: `collection.create_index([("location", "2dsphere")])`

Q: What is geofencing?
A: Creating a virtual perimeter around a geographic area to detect when points (e.g., devices, vehicles) enter or exit that area.

Q: How do you enable PostGIS in PostgreSQL?
A: `CREATE EXTENSION IF NOT EXISTS postgis;`

COMPARATIVE CONCEPTS ACROSS TOPICS

1. RELATIONAL (PostgreSQL) vs DOCUMENT (MongoDB)

Schema:

- PostgreSQL: Fixed schema, defined upfront
- MongoDB: Flexible schema, evolves with application

Relationships:

- PostgreSQL: Foreign keys with enforced constraints
- MongoDB: Manual references, no enforced constraints

Joins:

- PostgreSQL: Native JOIN operations
- MongoDB: Manual joins or $lookup aggregation

Transactions:

- PostgreSQL: ACID transactions, explicit commits
- MongoDB: Atomic operations on single documents

Scaling:

- PostgreSQL: Vertical scaling (more powerful server)
- MongoDB: Horizontal scaling (sharding across servers)

2. LOCAL vs CLOUD HOSTING

Local (Mongita, local PostgreSQL):

- Development and learning
- No network latency
- Full control
- Manual backups and maintenance

Cloud (Atlas, managed PostgreSQL):

- Production deployment
- Automatic backups and monitoring
- Scalability and high availability
- Managed security and updates
- Pay for usage

3. BASIC QUERIES vs AGGREGATION

Basic Queries:

- Simple find operations
- Single collection
- Filter, sort, limit
- Manual data processing in application

Aggregation:

- Complex multi-stage pipelines

- Cross-collection operations ($lookup)
- Server-side data transformation
- More efficient for complex analysis

PRACTICAL EXAM PREPARATION

HANDS-ON EXERCISES:

1. **PostgreSQL:**
    - Set up a PostgreSQL database with two related tables
    - Create users with different permission levels
    - Write Python code to connect and perform CRUD operations
    - Convert SQLite code to PostgreSQL
2. **MongoDB Basics:**
    - Create collections and insert documents with varied structures
    - Practice CRUD operations with query operators ($gt, $set, etc.)
    - Understand cursor behavior by iterating twice
    - Convert ObjectId to/from strings
3. **MongoDB with Flask:**
    - Build a simple Flask app with MongoDB backend
    - Implement manual joins between collections
    - Handle ObjectId conversions
    - Test CRUD operations
4. **MongoDB Atlas:**
    - Connect to cloud-hosted MongoDB
    - Handle connection errors
    - Secure connection strings with environment variables
5. **Aggregation Pipeline:**
    - Write pipelines with $lookup, $unwind, $project
    - Create computed fields with $cond
    - Practice accessing nested fields
6. **Geospatial:**
    - Create GeoJSON point and polygon documents
    - Write geofencing queries with $geoIntersects
    - Find points within areas with $geoWithin
    - Use PostGIS functions in PostgreSQL

CODE PATTERNS TO MEMORIZE:

1. PostgreSQL connection:

```
connection = psycopg2.connect(
    host='localhost',
    database='db_name',
    user='username',
    password='password',
    cursor_factory=RealDictCursor
)
```

2. MongoDB CRUD:

```
collection.insert_one({...})
collection.find({'field': {'$gt': value}})
collection.update_one({'_id': id}, {'$set': {...}})
collection.delete_one({'_id': id})
```

3. Manual join in MongoDB:

```
for pet in pets:
    kind = kind_collection.find_one({'_id': pet['kind_id']})
    pet['kind_name'] = kind['kind_name']
```

4. Aggregation lookup:

```
collection.aggregate([
  {"$lookup": {
    "from": "other_collection",
    "localField": "local_field",
    "foreignField": "foreign_field",
    "as": "result_array"
  }},
  {"$unwind": "$result_array"}
])
```

5. Geofencing:

```
zones.find_one({
    "area": {
        "$geoIntersects": {
            "$geometry": {
                "type": "Point",
                "coordinates": [lon, lat]
            }
        }
```

```
    }
})
```

COMMON MISTAKES TO AVOID:

1. Using `?` instead of `%s` in PostgreSQL queries
2. Forgetting to `commit()` in PostgreSQL
3. Trying to iterate a MongoDB cursor twice
4. Not converting string to ObjectId before querying
5. Using latitude before longitude in GeoJSON
6. Hardcoding passwords in code
7. Forgetting to create geospatial indexes
8. Using `$set` without braces: `{'$set': data}` not `{'$set': 'value'}`
9. Comparing string IDs with ObjectId IDs
10. Not handling connection errors in cloud databases

KEY TERMINOLOGY:

- **ACID**: Atomicity, Consistency, Isolation, Durability
- **MVCC**: Multi-Version Concurrency Control
- **CRUD**: Create, Read, Update, Delete
- **BSON**: Binary JSON (MongoDB's storage format)
- **ObjectId**: Unique identifier in MongoDB
- **Cursor**: Iterator over query results
- **Aggregation Pipeline**: Series of data transformations
- **GeoJSON**: JSON format for geographic data
- **SRID**: Spatial Reference System Identifier
- **Geofencing**: Virtual geographic boundary

BEST PRACTICES CHECKLIST:

- □ Store credentials in environment variables or config files
- □ Create users with appropriate privileges
- □ Use parameterized queries (prevent SQL injection)
- □ Add .env files to .gitignore
- □ Create indexes on frequently queried fields
- □ Test database operations with unit tests
- □ Handle connection errors appropriately
- □ Use connection pooling for production applications

PERFORMANCE TIPS:

- □ Create indexes on frequently queried fields
- □ Use $match early in aggregation pipelines
- □ Use connection pooling for production
- □ Create geospatial indexes (2dsphere) for location queries
- □ Use aggregation pipelines instead of manual joins for large datasets
- □ Monitor query performance with EXPLAIN (PostgreSQL) or explain()

- □ Batch operations when possible (insert_many vs multiple insert_one)

FINAL REVIEW CHECKLIST

PostgreSQL (Topic 08):

- □ Installation and setup process
- □ User creation and permission management
- □ Differences from SQLite (SERIAL, %s, VARCHAR)
- □ psycopg2 connection and usage
- □ Foreign key constraints
- □ Backup and restore procedures

MongoDB Concepts (Topic 09):

- □ Document vs relational paradigm
- □ CRUD operations (insert, find, update, delete)
- □ Query operators ($gt, $set, etc.)
- □ Cursor behavior and consumption
- □ ObjectId purpose and usage
- □ Schema flexibility
- □ Mongita vs MongoDB

Flask with MongoDB (Topic 10):

- □ Application architecture
- □ Manual joins between collections
- □ ObjectId string conversion
- □ CRUD implementation patterns
- □ Referential integrity handling
- □ Testing patterns

MongoDB Atlas (Topic 11):

- □ Cloud database concepts
- □ Connection strings and authentication
- □ Security best practices
- □ Error handling
- □ Migration from local to cloud

Aggregation (Topic 12):

- □ Pipeline concept and stages
- □ $lookup for joins
- □ $unwind for arrays
- □ $project for field selection
- □ $match for filtering
- □ Conditional expressions ($cond)

- □ Accessing nested fields

Geospatial (Topic 13):

- □ GeoJSON format and coordinate order
- □ Geospatial indexes (2dsphere)
- □ $geoIntersects for geofencing
- □ $geoWithin for area queries
- □ PostGIS extension for PostgreSQL
- □ Common geospatial use cases

GOOD LUCK ON YOUR EXAM!

Remember:

- Understand concepts, don't just memorize syntax
- Practice writing code by hand
- Draw diagrams for complex queries and pipelines
- Think about use cases: when to use each technology
- Review error messages and troubleshooting steps
- Always use parameterized queries to prevent SQL injection

> "The expert in anything was once a beginner." - Helen Hayes