통합

1. install

여기서는 macOS에서 MongoDB를 설치하고 관리하는 방법을 설명합니다. 아래는 각 명령어의 역할과 설명입니다:

1. Xcode 명령어 도구 설치

```
xcode-select --install
```

- **목적**: Xcode 명령어 도구를 설치합니다. 이 도구는 macOS에서 소프트웨어를 컴파일하고 설치하는 데 필요한 유틸리티를 포함하고 있습니다. Homebrew와 같은 도구를 사용하려면 이 도구가 필요합니다.

2. Homebrew에 MongoDB 저장소 추가

```
brew tap mongodb/brew
```

- **목적**: MongoDB의 공식 Homebrew 저장소를 추가합니다. 이 저장소는 MongoDB를 설치할 때 사용되는 패키지와 포뮬러를 포함하고 있습니다.

3. Homebrew 업데이트

```
brew update
```

- **목적**: Homebrew와 패키지 목록을 최신 상태로 업데이트합니다. 이를 통해 최신 버전의 소프트웨어를 사용할 수 있습니다.

4. MongoDB Community Edition 설치

```
brew install mongodb-community@7.0
```

- **목적**: Homebrew를 사용하여 MongoDB Community Edition 7.0 버전을 설치합니다. 이 버전은 `mongodb/brew` 저장소에서 제공하는 최신 버전입니다.

5. MongoDB 서비스 시작

```
brew services start mongodb-community@7.0
```

- **목적**: MongoDB 서비스를 시작합니다. 이 명령어를 사용하면 MongoDB가 백그라운드에서 실행되며, 시스템 부팅 시 자동으로 시작되도록 설정됩니다.

6. MongoDB 서비스 중지

```
brew services stop mongodb-community@7.0
```

- **목적**: MongoDB 서비스를 중지합니다.

7. Intel 기반 Mac에서 MongoDB 서비스 확인

```
mongod --config /usr/local/etc/mongod.conf --fork
```

- **목적**: Intel 기반 Mac에서 MongoDB를 백그라운드 프로세스로 시작합니다. `--fork` 옵션을 사용하여 MongoDB가 백그라운드에서 실행되도록 합니다.

8. Apple Silicon Mac에서 MongoDB 서비스 확인

```
mongod --config /opt/homebrew/etc/mongod.conf --fork
```

- **목적**: Apple Silicon Mac (M1/M2 칩)에서 MongoDB를 백그라운드 프로세스로 시작합니다. Apple Silicon 시스템에서는 설정 파일의 경로가 다릅니다.

9. Homebrew 서비스 목록 확인

```
brew services list
```

- **목적**: Homebrew에서 관리하는 모든 서비스의 목록을 표시하고, 각 서비스의 현재 상태(시작됨, 중지됨 등)를 확인합니다.

10. MongoDB 프로세스 확인

```
ps aux | grep -v grep | grep mongod
```

- **목적**: `mongod`와 관련된 실행 중인 프로세스를 확인합니다. `grep -v grep`은 `grep` 명령어 자체를 결과에서 제외합니다.

11. MongoDB 쉘 실행

```
mongosh
```

- **목적**: MongoDB 쉘(`mongosh`)을 실행하여 MongoDB 인스턴스와 상호작용할 수 있습니다.

요약

- **설치**: 필요한 도구를 설치한 후 Homebrew를 사용하여 MongoDB를 설치합니다.
- **서비스 관리**: MongoDB 서비스를 시작하거나 중지하고 상태를 확인하는 명령어를 사용합니다.
- **구성 및 모니터링**: MongoDB를 수동으로 시작하고 프로세스를 모니터링하여 MongoDB가 제대로 실행되고 있는지 확인합니다.

이 절차를 통해 macOS에서 MongoDB를 올바르게 설치하고 관리할 수 있습니다.

2. insert

이 MongoDB 쿼리들은 `inventory` 컬렉션에 문서를 삽입하고, 삽입한 문서를 검색하는 방법을 보여줍니다. 각 쿼리에 대한 설명은 다음과 같습니다:

1. 단일 문서 삽입

```
db.inventory.insertOne(
   { item: "canvas", qty: 100, tags: ["cotton"], size: { h: 28, w: 35.5,
uom: "cm" } }
)
```

- **설명**: `inventory` 컬렉션에 단일 문서를 삽입합니다.
  - 문서의 내용:
    - `item` : "canvas"
    - `qty` : 100
    - `tags` : 배열로 `["cotton"]` 포함
    - `size` : 객체로 `h` (높이), `w` (너비), `uom` (단위)를 포함

2. 문서 검색 (단일 문서)

```
db.inventory.find( { item: "canvas" } )
```

- **설명**: `item` 필드가 `"canvas"` 인 문서를 검색합니다.
  - 이 쿼리는 `item` 필드가 `"canvas"` 인 모든 문서를 반환합니다.

3. 여러 문서 삽입

```
db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w:
21, uom: "cm" } },
   { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom:
"cm" } },
   { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w:
```

```
22.85, uom: "cm" } }
])
```

- **설명**: `inventory` 컬렉션에 여러 문서를 동시에 삽입합니다.
  - 문서의 내용:
    - 첫 번째 문서:
      - `item` : "journal"
      - `qty` : 25
      - `tags` : 배열로 `["blank", "red"]` 포함
      - `size` : 객체로 `h`, `w`, `uom` 포함
    - 두 번째 문서:
      - `item` : "mat"
      - `qty` : 85
      - `tags` : 배열로 `["gray"]` 포함
      - `size` : 객체로 `h`, `w`, `uom` 포함
    - 세 번째 문서:
      - `item` : "mousepad"
      - `qty` : 25
      - `tags` : 배열로 `["gel", "blue"]` 포함
      - `size` : 객체로 `h`, `w`, `uom` 포함

4. 문서 검색 (전체 컬렉션)

```
db.inventory.find( {} )
```

- **설명**: `inventory` 컬렉션의 모든 문서를 검색합니다.
  - `{}` 빈 쿼리 객체를 사용하여 컬렉션에 있는 모든 문서를 반환합니다.

이 쿼리들은 MongoDB에서 데이터를 삽입하고 검색하는 기본적인 작업을 다루고 있으며, 컬렉션에 문서를 추가하고, 특정 조건으로 검색하며, 전체 문서를 조회하는 방법을 보여줍니다.

3. query

이 MongoDB 쿼리들은 다양한 검색 및 데이터 조작 작업을 수행하는 방법을 보여줍니다. 아래는 각 쿼리의 설명입니다:

1. 문서 삽입

```
db.inventory.insertMany([
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status:
"A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" },
```

```
status: "A" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status:
"D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" },
status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" },
status: "A" }
]);
```

- **목적**: `inventory` 컬렉션에 여러 문서를 한 번에 삽입합니다. 각 문서는 `item`, `qty`, `size`, `status` 필드를 포함하고 있습니다.

2. 컬렉션에서 모든 문서 선택

```
db.inventory.find( {} )
```

- **목적**: `inventory` 컬렉션의 모든 문서를 반환합니다.

3. 동등성 조건 지정

```
db.inventory.find( { status: "D" } )
```

- **목적**: `status` 필드가 `"D"` 인 문서만 반환합니다.

4. 쿼리 연산자를 사용하여 조건 지정

```
db.inventory.find( { status: { $in: [ "A", "D" ] } } )
```

- **목적**: `status` 필드가 `"A"` 또는 `"D"` 인 문서만 반환합니다. `$in` 연산자는 배열에 포함된 값 중 하나와 일치하는 문서를 찾습니다.

5. AND 조건 지정

```
db.inventory.find( { status: "A", qty: { $lt: 30 } } )
```

- **목적**: `status` 가 `"A"` 이고 `qty` 가 30 미만인 문서만 반환합니다.

6. OR 조건 지정

```
db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

- **목적**: `status` 가 `"A"` 이거나 `qty` 가 30 미만인 문서만 반환합니다. `$or` 연산자는 배열 내의 조건 중 하나라도 일치하는 문서를 찾습니다.

7. AND와 OR 조건 지정

```
db.inventory.find( {
     status: "A",
     $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]
} )
```

- **목적**: `status`가 `"A"`이고, `qty`가 30 미만이거나 `item`이 `"p"`로 시작하는 문서만 반환합니다. `$or` 내의 두 조건 중 하나가 맞으면 문서가 반환됩니다.

8. 점 표기법을 사용하는 중첩 필드 쿼리

```
db.inventory.find( { "size.uom": "in" } )
```

- **목적**: `size.uom` 필드가 `"in"`인 문서만 반환합니다.

```
db.inventory.find( { "size.h": { $lt: 15 } } )
```

- **목적**: `size.h` 필드가 15 미만인 문서만 반환합니다.

9. 배열 쿼리

```
db.inventory.find( { tags: ["red", "blank"] } )
```

- **목적**: `tags` 배열이 정확히 `["red", "blank"]`인 문서만 반환합니다.

```
db.inventory.find( { tags: { $all: ["red", "blank"] } } )
```

- **목적**: `tags` 배열에 `"red"`와 `"blank"`가 모두 포함된 문서만 반환합니다. 배열에 특정 값이 포함되어야 함을 검사합니다.

10. 배열 요소에 대한 쿼리

```
db.inventory.find( { tags: "red" } )
```

- **목적**: `tags` 배열에 `"red"`가 포함된 문서만 반환합니다.

```
db.inventory.find( { dim_cm: { $gt: 25 } } )
```

- **목적**: `dim_cm` 배열에 25보다 큰 값이 포함된 문서만 반환합니다.

11. 임베디드 문서 배열 쿼리

```
db.inventory.find( { "instock": { qty: 5, warehouse: "A" } } )
```

- 목적: `instock` 배열에 `qty`가 5이고 `warehouse`가 `"A"`인 문서만 반환합니다.

```
db.inventory.find( { 'instock.qty': { $lte: 20 } } )
```

- 목적: `instock` 배열의 `qty` 필드가 20 이하인 문서만 반환합니다.

12. 문서 배열에 여러 조건 지정

```
db.inventory.find( { "instock": { $elemMatch: { qty: 5, warehouse: "A" } }
} )
```

- 목적: `instock` 배열의 요소 중 `qty`가 5이고 `warehouse`가 `"A"`인 문서만 반환합니다.

13. 프로젝션

```
db.inventory.find( { status: "A" }, { item: 1, status: 1 } )
```

- 목적: `status`가 `"A"`인 문서에서 `item`과 `status` 필드만 반환합니다. `_id` 필드는 기본적으로 포함됩니다.

```
db.inventory.find( { status: "A" }, { item: 1, status: 1, _id: 0 } )
```

- 목적: `status`가 `"A"`인 문서에서 `item`과 `status` 필드만 반환하며 `_id` 필드는 제외합니다.

14. 스위치와 문자열 결합

```
db.inventory.find(
   { },
   {
      _id: 0,
      item: 1,
      status: {
         $switch: {
            branches: [
               {
                  case: { $eq: [ "$status", "A" ] },
                  then: "Available"
               },
               {
                  case: { $eq: [ "$status", "D" ] },
                  then: "Discontinued"
               },
```

```
            ],
            default: "No status found"
         }
      },
      area: {
         $concat: [
            { $toString: { $multiply: [ "$size.h", "$size.w" ] } },
            " ",
            "$size.uom"
         ]
      },
      reportNumber: { $literal: 1 }
   }
)
```

- **목적**: 문서에서 `status`에 따라 `status`를 변환하거나, `size.h`와 `size.w`를 곱해 면적을 계산하고, 문자열을 결합하여 `area`를 생성하며, `reportNumber`를 1로 설정합니다.

15. `null` 필드

```
db.inventory.insertMany([
   { _id: 1, item: null },
   { _id: 2 }
])
```

- **목적**: `item`이 `null`인 문서와 `item` 필드가 없는 문서를 삽입합니다.

```
db.inventory.find( { item: null } )
```

- **목적**: `item` 필드가 `null`인 문서를 찾습니다.

```
db.inventory.find( { item: { $ne : null } } )
```

- **목적**: `item` 필드가 `null`이 아닌 문서를 찾습니다.

```
db.inventory.find( { item : { $exists: false } } )
```

- **목적**: `item` 필드가 존재하지 않는 문서를 찾습니다.

이 쿼리들은 MongoDB에서 다양한 방식으로 데이터를 쿼리하고 조작하는 방법을 보여줍니다. 데이터의 필터링, 정렬, 조건부 검색, 배열 및 중첩 문서 쿼리, 프로젝션 등 여러 기능을 활용할 수 있습니다.

3. update

이 MongoDB 쿼리들은 다양한 `update`, `replace`, `aggregate` 연산을 보여줍니다. 각 섹션에 대해 자세히 설명하겠습니다.

---

1. 단일 문서 업데이트

```
db.inventory.updateOne(
   { item: "paper" },
   {
     $set: { "size.uom": "cm", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

- **설명**: `item` 필드가 `"paper"`인 문서를 하나 찾아서 업데이트합니다.
  - `size.uom`을 `"cm"`로 변경하고, `status`를 `"P"`로 설정합니다.
  - 문서의 `lastModified` 필드를 현재 날짜로 업데이트합니다.

2. 여러 문서 업데이트

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

- **설명**: `qty` 필드가 50 미만인 모든 문서를 업데이트합니다.
  - `size.uom`을 `"in"`로, `status`를 `"P"`로 설정합니다.
  - `lastModified` 필드를 현재 날짜로 설정합니다.

3. 문서 교체

```
db.inventory.replaceOne(
   { item: "paper" },
   { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse:
"B", qty: 40 } ] }
)
```

- **설명**: `item` 필드가 `"paper"`인 문서를 찾아 전체 문서를 새 문서로 교체합니다.
  - 기존 문서를 삭제하고, `item: "paper"`와 `instock` 배열이 포함된 새 문서를 삽입합니다.

4. 집계 파이프라인 - 단일 필드 업데이트

```
db.students.updateOne( { _id: 3 }, [ { $set: { "test3": 98, modified:
"$$NOW"} } ] )
```

- **설명**: `_id`가 `3`인 문서의 `test3` 필드를 `98`로 설정하고, `modified` 필드를 현재 날짜로 설정합니다.
  - `$set`과 함께 집계 파이프라인이 사용되어 더 복잡한 연산이 가능합니다.

5. 집계 파이프라인 - 여러 필드 업데이트

```
db.students2.updateMany( {},
  [
    { $replaceRoot: { newRoot:
       { $mergeObjects: [ { quiz1: 0, quiz2: 0, test1: 0, test2: 0 },
"$$ROOT" ] }
    } },
    { $set: { modified: "$$NOW"}  }
  ]
)
```

- **설명**: 모든 문서를 대상으로 `quiz1`, `quiz2`, `test1`, `test2` 필드를 추가하거나 기본값으로 설정합니다. 기존 문서 내용은 유지하면서 새로운 필드를 병합합니다.
  - `modified` 필드를 현재 날짜로 설정합니다.

6. 집계 파이프라인 - 평균 및 등급 계산

```
db.students3.updateMany(
   { },
   [
     { $set: { average : { $trunc: [ { $avg: "$tests" }, 0 ] }, modified:
"$$NOW" } },
     { $set: { grade: { $switch: {
                           branches: [
                               { case: { $gte: [ "$average", 90 ] }, then:
"A" },
                               { case: { $gte: [ "$average", 80 ] }, then:
"B" },
                               { case: { $gte: [ "$average", 70 ] }, then:
"C" },
                               { case: { $gte: [ "$average", 60 ] }, then:
"D" }
                           ],
                           default: "F"
     } } } }
```

```
    ]
)
```

- **설명**: 각 학생 문서에서 `tests` 배열의 평균 점수를 계산하고, 이를 `average` 필드에 저장합니다.
  - `average` 값을 기반으로 등급을 계산하고 `grade` 필드에 저장합니다.
  - `modified` 필드를 현재 날짜로 설정합니다.

7. 배열 필드 업데이트

```
db.students4.updateOne( { _id: 2 },
  [ { $set: { quizzes: { $concatArrays: [ "$quizzes", [ 8, 6 ]  ] } } } ]
)
```

- **설명**: `_id`가 `2`인 문서의 `quizzes` 배열에 `[8, 6]`을 추가합니다.
  - `concatArrays`를 사용하여 기존 배열에 새로운 요소를 결합합니다.

8. $addFields가 포함된 updateMany

```
db.temperatures.updateMany( { },
  [
    { $addFields: { "tempsF": {
          $map: {
             input: "$tempsC",
             as: "celsius",
             in: { $add: [ { $multiply: ["$$celsius", 9/5 ] }, 32 ] }
          }
    } } }
  ]
)
```

- **설명**: 모든 문서에 대해 `tempsC` 배열에 있는 값을 화씨(`Fahrenheit`)로 변환하여 `tempsF` 필드를 추가합니다.
  - 각 섭씨 값을 화씨로 변환하기 위해 `$map`과 `$add` 연산을 사용합니다.

9. $let 변수를 사용한 업데이트

```
db.cakeFlavors.updateOne(
   {
      $expr: { $eq: [ "$flavor", "$$targetFlavor" ] }
   },
   [
      {
         $set: { flavor: "$$newFlavor" }
      }
   ],
```

```
    {
        let: { targetFlavor: "cherry", newFlavor: "orange" }
    }
)
```

- **설명**: `flavor`가 `"cherry"`인 문서를 찾아 `flavor` 필드를 `"orange"`로 업데이트합니다.
    - `$let` 변수를 사용하여 조건과 업데이트 값을 동적으로 설정합니다.

이 쿼리들은 MongoDB에서 다양한 방식으로 데이터를 업데이트, 교체, 또는 집계하는 방법을 보여주며, 특히 집계 파이프라인을 사용하여 복잡한 업데이트 작업을 수행하는 방법을 강조합니다.

4. delete

MongoDB에서 데이터 삭제를 수행하는 쿼리들에 대한 설명은 다음과 같습니다:

1. 문서 삽입

```
db.inventory.insertMany( [
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status:
"A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" },
status: "P" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status:
"D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" },
status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" },
status: "A" }
] );
```

- **목적**: `inventory` 컬렉션에 5개의 문서를 삽입합니다. 각 문서는 `item`, `qty`, `size`, `status` 필드를 포함하고 있습니다. 이 문서들은 다양한 `status` 값을 가지고 있습니다: `"A"`, `"P"`, `"D"`.

2. `status`가 `"A"`인 문서 삭제

```
db.inventory.deleteMany({ status: "A" })
```

- **목적**: `status`가 `"A"`인 모든 문서를 삭제합니다. 이 쿼리는 여러 문서가 해당 조건을 만족할 경우 모두 삭제합니다.

3. `status`가 `"D"`인 문서 삭제

```
db.inventory.deleteOne({ status: "D" })
```

- **목적**: `status`가 `"D"`인 문서 중 첫 번째 문서 하나를 삭제합니다. `deleteOne`은 조건에 맞는 문서가 여러 개일지라도 첫 번째로 찾은 하나의 문서만 삭제합니다.

4. 모든 문서 삭제

```
db.inventory.deleteMany({})
```

- **목적**: `inventory` 컬렉션의 모든 문서를 삭제합니다. 조건 없이 모든 문서를 삭제하므로 컬렉션이 비워집니다.

이러한 삭제 작업들은 데이터베이스의 상태를 변경하며, 실수로 중요한 데이터를 삭제하지 않도록 주의해서 사용해야 합니다. 일반적으로 삭제 작업을 수행하기 전에 `find` 쿼리를 통해 삭제될 문서들을 검토하는 것이 좋습니다.

5. mongo vs rdb

MongoDB와 같은 NoSQL 데이터베이스는 관계형 데이터베이스(RDB)와 다른 구조를 가지며, 특정한 상황에서 더 유리할 수 있습니다. MongoDB는 문서 지향(Document-Oriented) 데이터베이스로, JSON 형식의 문서를 저장하고 처리하는 데 최적화되어 있습니다. MongoDB를 사용하는 것이 좋은 경우와 RDB와 비교한 차이점은 다음과 같습니다.

1. MongoDB를 사용하는 것이 좋은 경우

1. **비정형 데이터(유연한 스키마) 처리**
   - **데이터 구조가 유동적이거나 미리 정의된 스키마가 없는 경우** MongoDB가 유리합니다. 관계형 DB는 미리 정의된 스키마를 따르지만, MongoDB는 스키마가 고정되지 않아 각 문서가 서로 다른 필드와 구조를 가질 수 있습니다.
   - 예: 소셜 미디어 플랫폼에서 사용자 프로필이 각기 다른 필드와 정보를 가질 수 있을 때 MongoDB가 더 적합합니다.
2. **복잡한 관계가 적은 경우**
   - MongoDB는 관계형 데이터베이스처럼 명시적인 외래 키를 사용하지 않기 때문에, **데이터 간의 관계가 복잡하지 않거나 조인(Join)이 자주 필요하지 않은** 경우에 적합합니다.
   - 예: 로그 데이터나 이벤트 데이터를 다룰 때, 각 이벤트가 독립적으로 저장되고 다른 테이블과의 조인이 거의 필요 없는 경우.
3. **수평적 확장이 필요한 경우**
   - MongoDB는 **수평적 확장성(Scalability)**을 고려하여 설계되었습니다. 대규모 데이터를 분산된 여러 서버에 저장하고 관리하는 데 효율적입니다.
   - 관계형 데이터베이스는 수직적 확장(더 좋은 서버로 업그레이드)을 선호하는 반면, MongoDB는 **샤딩(Sharding)**을 통해 데이터가 자동으로 여러 서버에 분산됩니다.
   - 예: 대규모 사용자 데이터를 관리하거나 글로벌 분산 시스템에서 성능을 유지해야 할 때

MongoDB가 유리합니다.

4. **고속 읽기/쓰기 작업이 필요한 경우**
   - MongoDB는 데이터의 **고속 쓰기 작업**에 최적화되어 있습니다. 특히 트랜잭션의 복잡성이 낮고, 실시간으로 대용량 데이터를 입력해야 하는 시스템에서 좋은 성능을 발휘합니다.
   - 예: 실시간 분석, 로그 수집, IoT 데이터 처리 등에서 빠른 데이터를 수집하고 저장할 때 MongoDB가 유리합니다.
5. **데이터 중첩이 많은 경우**
   - **내장 문서(Embedded Documents)** 구조를 지원하기 때문에 중첩된 데이터를 쉽게 저장할 수 있습니다. 관계형 DB에서는 이를 정규화해야 하지만, MongoDB는 중첩된 구조의 데이터를 직관적으로 저장할 수 있어 복잡한 구조의 데이터를 쉽게 처리할 수 있습니다.
   - 예: 제품 정보에 여러 옵션(사이즈, 색상 등)이 있는 쇼핑몰 시스템에서 각 제품에 옵션을 중첩 구조로 저장할 때.
6. **빅데이터와 실시간 분석**
   - MongoDB는 데이터의 대규모 처리 및 실시간 분석에 적합합니다. 관계형 데이터베이스는 데이터를 정규화하고 스키마에 맞춰야 하므로 대용량 데이터를 다룰 때 오버헤드가 발생할 수 있지만, MongoDB는 **유연한 스키마와 빠른 읽기/쓰기 성능** 덕분에 빅데이터 처리에서 장점을 가집니다.

2. MongoDB가 적합하지 않은 경우

1. **복잡한 트랜잭션이 필요한 경우**
   - 관계형 데이터베이스는 **ACID**(Atomicity, Consistency, Isolation, Durability) 트랜잭션을 보장합니다. 은행 시스템처럼 여러 테이블에서의 트랜잭션 일관성을 보장해야 하는 경우 RDB가 적합합니다.
   - MongoDB는 간단한 트랜잭션에는 적합하지만, **복잡한 트랜잭션** 처리에는 관계형 DB만큼 강력하지 않습니다.
2. **데이터 간의 관계가 복잡한 경우**
   - 데이터 간 **다대다(N:M)** 관계나 **조인(Join)**이 자주 발생하는 경우 MongoDB는 비효율적일 수 있습니다. 관계형 데이터베이스는 조인을 통해 여러 테이블에서 데이터를 결합하는 데 특화되어 있지만, MongoDB는 조인을 지원하지 않거나 비효율적입니다.
3. **엄격한 스키마가 필요한 경우**
   - 데이터가 **명확하고 일관된 스키마**를 가져야 하는 경우 관계형 데이터베이스가 적합합니다. MongoDB는 스키마 유연성이 있지만, 이를 관리하지 않으면 데이터의 일관성을 유지하기 어려울 수 있습니다.

3. MongoDB와 RDB 비교 요약

특징	**MongoDB**	관계형 데이터베이스(RDB)
데이터 구조	유연한 스키마 (비정형)	고정된 스키마 (정형)
관계 처리	중첩 문서로 처리	조인으로 처리
확장성	수평적 확장 (샤딩)	수직적 확장
트랜잭션	단순 트랜잭션 지원	복잡한 ACID 트랜잭션

특징	**MongoDB**	관계형 데이터베이스(**RDB**)
읽기/쓰기 성능	고속	다소 느릴 수 있음
사용 사례	비정형 데이터, 빅데이터, 로그, 실시간 분석	복잡한 트랜잭션, 관계가 복잡한 데이터

MongoDB는 유연성, 확장성, 속도 측면에서 유리한 반면, 관계형 데이터베이스는 트랜잭션 일관성과 데이터 관계 관리에서 강점을 가집니다. 따라서 프로젝트의 요구사항에 따라 두 데이터베이스 중 적합한 것을 선택해야 합니다.