MongoDB Relationship & Modeling

소프트웨어공학과 / 201332001 강인성 / hogu9401@gmail.com

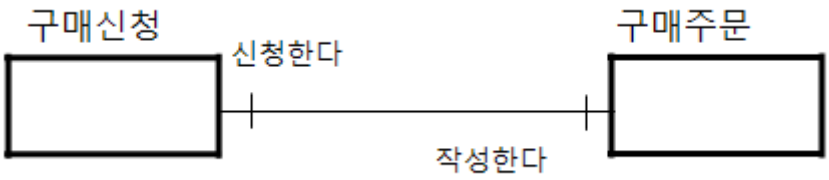
1. RDBMS에서의 Relationship & Modeling

A. RDBMS에서 관계의 종류

Relational Database에서의 관계는 Foreign Key(외래 키)를 이용해서 형성을 해야 한다. 데이터베이스 개념에 대해 복습을 하는 기회를 가져서 이에 대해서 알아본다면 Relational Database의 관계의 종류는 1:1, 1:N, M:N 3가지로 나뉠 수 있다.

➔ 1 : 1 관계

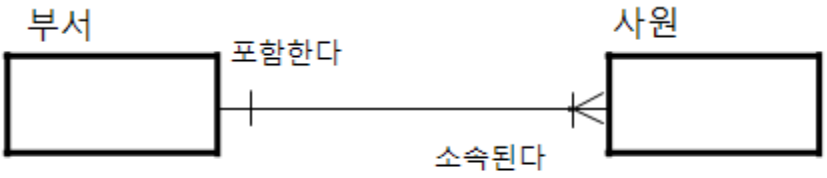
어느 쪽 Entity의 입장에서도 타 Entity의 상대를 봐도 반드시 단 하나의 관계를 형성한다고 생각하면 된다. 예를 들어 부모 관계, 연인 관계 등으로 나뉘어 볼 수 있다.



[그림] 1:1 관계를 ER 다이어그램으로 이처럼 그릴 수 있다.

➔ 1 : N 관계

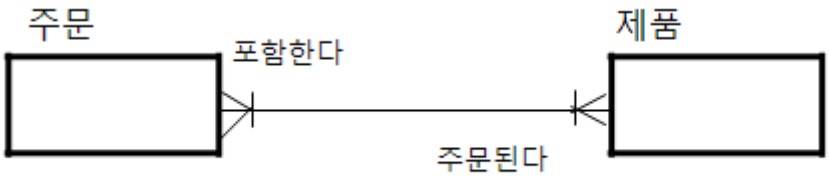
한 쪽이 관계를 맺은 쪽의 여러 객체를 갖는 것을 의미를 한다. 여기서는 부모 Entity와 자식 Entity로 나뉘어 설명을 한다면 대부분 부모 Entity에 대한 데이터 로딩을 즉시 불러오고(Eager), 자식 Entity에 대해서는 추후에 불러오는(Lazy) 경향을 가지고 있다. 1 : N 관계에 대해서는 고급 웹 프로그래밍1, 2에서 구현 시험 때 공부를 하면서 많이 경험을 해 봤기 때문에 이에 대해서는 크게 설명하지 않고 넘어가겠다.



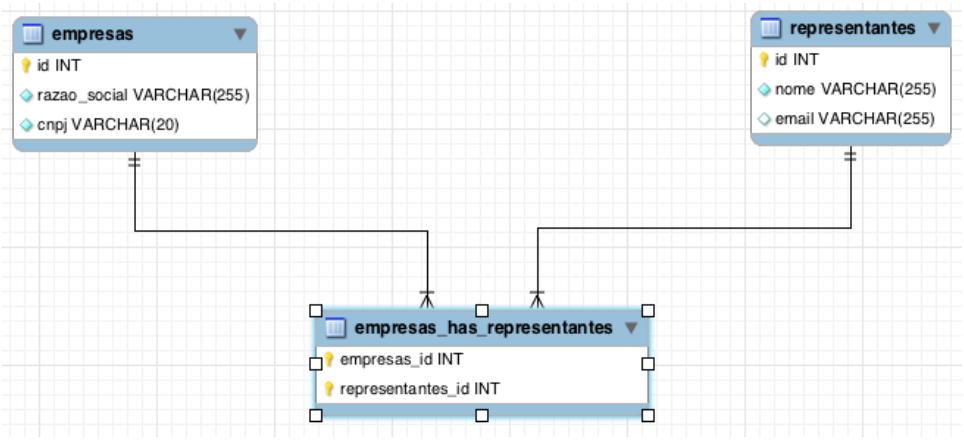
[그림] 1:N 관계를 ER 다이어그램으로 이처럼 그릴 수 있다.

➔ M : N 관계

관계를 가진 양쪽의 Entity에게 1 : N 관계가 존재할 때 나타내는 관계로 볼 수 있는데 이를 실제로 SQL에서 구현을 하기 위해서는 중앙에 양쪽 Entity에 존재하는 Primary Key(기본 키)들을 통한 M:N 관계를 저장하는 테이블을 하나 더 만들어서 작동을 시키는 경우가 대다수이다. M : N 관계의 사례로는 각 학생이 여러 과목들을 수강을 하는 경우에 대해서 사례를 들 수 있다.



[그림] M : N 관계를 ER 다이어그램으로 이처럼 그릴 수 있다.



[그림] 그렇지만 SQL 문에서는 M : N 관계에 적합한 모델링은 이 밖에 없지만 JPA에서는 @ManyToMany를 기반으로 객체 모델링을 하는데 다행히 지장은 없다.

B. NoSQL와의 차이점

Relational Database와 MongoDB의 데이터 모델링에는 미묘한 차이가 보인다. 우리가 관계형 데이터베이스를 이용해서 스키마를 구상할 때에는 테이블 디자인(논리적 모델링 과정)을 먼저 하고 난 후에 쿼리 디자인(물리적 모델링 과정)을 거쳤다. 이를 Entity 모델 지향으로 볼 수 있는데 Relational Database에서는 쿼리를 어떻게 복잡하게 작성을 해도 지원을 해서 테이블 간의 중복을 최소화를 위해 정규화 작업을 진행을 하는 것과는 달리 NoSQL(MongoDB)에서는 어플리케이션의 데이터 접근 패턴을 고려를 하여 매우 단순한 쿼리를 위한 Document를 디자인을 할 필요가 있는데 이를 Query 모델 지향으로 칭한다. NoSQL에서는 정규화를 하는 것과 달리 Document에서 데이터가 2개 이상 Document에 중복 저장을 어느 정도 허가하는 비정규화 테이블을 설계하는 경우(Embedded Document와 같은 개념)가 있다.

방금 전에 설명한 내용들을 토대로 한다면 아래와 같이 정리할 수 있다.

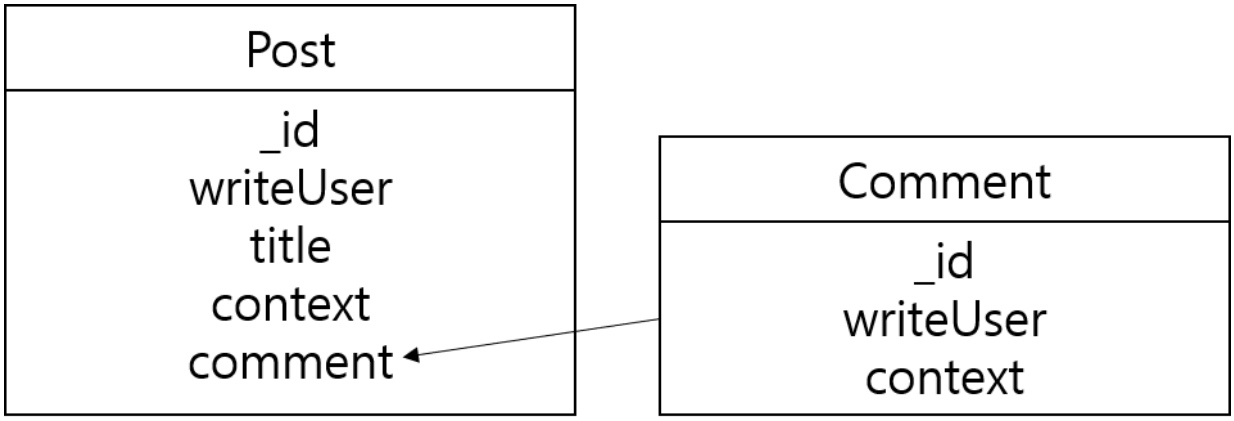
RDBMS	NoSQL
모델 디자인 후 쿼리 디자인	쿼리 디자인 후 모델 디자인
Entity 모델 지향	Query 모델 지향
데이터 중복 최소화	데이터 중복 허용
정규화 작업 위주	비정규화 작업 위주

2. MongoDB Relationship

NoSQL에서는 방금 전에서 언급을 했듯이 Query를 기반으로 하는 데이터 모델링을 추구를 하고 있는데 다행이 MongoDB에서는 관계를 형성해서 다른 Document를 참조해서 큰 Document를 완성하는 기능을 제공하고 있다. 이것이 NoSQL의 종류들 중에서 Key-Document 모델이 제공을 하는 기능인데 이번에는 MongoDB에서 관계를 조성하는 Embedded Document와 Reference Document 개념에 대해서 자세하게 알아보도록 하겠다. 참고로 처음에 이 내용들이 생각보다 까다로운 이야기들이 많기 때문에 필자는 관계형 데이터베이스에서 향시에 다루는 1 : N 관계를 기반으로 더욱 알기 쉽게 설명을 하겠다.

A. Embedded Document

MongoDB에서 관계를 형성할 때에는 이를 추구를 많이 하고 있는 편이다. 우리가 부모 Document와 자식 Document의 시점을 참고를 해 본다면 우리가 관계형 데이터베이스에서는 오로지 부모 Document에 외래키를 두어 자식 Document를 참고해서 부모 Document 내에서는 정작 참고를 해야 볼 수 있는데 이는 반대로 부모 Document 내부에 자식 Document를 _id인 ObjectId을 제외한 상태에서 Document 객체 그 자체를 넣어서 내장된 Document를 쉽게 가져올 수 있도록 하는 개념이다. 예를 들어 아래와 같은 Document들이 존재한다고 구상을 해보자.



[그림] 게시글(Post)와 댓글(Comment)의 관계 형성을 RDBMS의 관점으로 구상을 한 모델링 이는 Reference Document에서도 다시 언급을 할 것이다.

이처럼 모델링을 한다고 가정을 한다면 아래와 같이 작성을 할 수가 있다.

```
[{
   _id : "~~~~",
   writeUser : "유저1",
   title : "게시판 글",
   context : "게시판 내용",
   comment : [{ writeUser : "유저2", context : "댓글1" },
                  { writeUser : "유저3", context : "댓글2" }]
```

```
> db.bbs01.post.find()
```

그렇지만 Document 내부에 또 다른 Document를 넣는 것을 따져 생각을 해 본다면 무조건 Document들이 꼭 내장 되어야 하는 것은 항상 올바른 방법은 아니다. Relational Database에서는 1:N 관계에 대해서는 그저 외래키만 작성만 잘 해서 쓰면 되는데 라는 생각을 했지만 MongoDB에서의 관계를 구상을 할 때에는 Document 내부에 포함을 시키는가에 대한 의문을 가지는 것이 대부분이다. 예를 들어 사람의 신상 정보를 작성할 때에 대해서 생각을 해 봐야 한다. 한 사람의 연락처는 집 전화, 휴대폰 전화, 팩스 전화 등을 해서 적어도 1~2개의 연락처가 존재하기 마련이다. 이러한 경우(One to Few)에는 아예 새로운 Document를 만들어서 주입하는 것보다는 Document에 배열로 정리를 하는 방안이 효율적이다.

```
{
   _id : "----",
   name : "사람",
   age : 20,
   address : "서울특별시 어디구 저기동 요기대로",
   callPhone : [ {type : "cell", number : "010-1234-5678"},
                 {type : "home", number : "02-123-4567"} ]
}
```

One To Few를 기반으로 한 Embedded Document를 작성한다면 위와 같이 작성을 할 수 있지만 그렇지만 이에 대한 단점도 있다면 독자적인 Document를 통해 Entity를 생성할 수 없다는 점이 있긴 하다. 그래서 One To Few를 기반으로 작성을 하는 경우에는 각 Entity의 세부 사항 등 검색을 그렇게 까지 많이 안 하는 Embedded Document로 꾸며 나가는 것이 좋은 선택이다.

B. Reference Document(갱신을 자주 할 때)

위에서는 부모 Document 내부에 자식 Document 객체를 아예 집어 넣어서 오로지 부모의 Document만을 이용해서 조회를 해 오는 것이 관건이었는데 MongoDB에서는 NoSQL에서 포함된 Document만으로는 역부족이어서 이에 대해서 보안을 하기 위하여 Reference Document라는 개념을 도입을 하였다. Reference Document는 방금 위에서 작성한 내용들에 대해서 일부 수정을 한다면 아래와 같이 작성을 하여 참고를 한다.

```
[{
   _id : "0001",
   writeUser : "유저1",
   title : "게시판 글",
   context : "게시판 내용",
   comment : [ ObjectId("A001"), ObjectId("A002") ]
}]
```

```
> db.bbs01.post.find()
```

```
[{ _id : "A001", writeUser : "유저2", context : "댓글1" }, { _id : "A002", writeUser : "유저3", context : "댓글2" }]
```

```
> db.bbs01.comment.find()
```

이를 살펴본다면 우리가 흔히 쓰는 SQL의 외래 참조와 같은 개념으로 짚어볼 수 있다. 또한 MongoDB에서도 배열 자체 내부에 오로지 자식의 Primary Key(여기서 부모 Document는 post, 자식 Document는 comment이다.)를

저장을 하여 작성을 해도 문제가 없기 때문에 1 : N 뿐만이 아니라 M : N에서도 대응하기 효율적으로 볼 수 있다.(반대로 MongoDB에서는 1 : 1를 대부분 BSON Recursion(재귀) 문제를 방지하고자 하기 위해 방금 전에 언급했던 Embedded Document를 권장하고 있다.)

그래서 예를 들어 위에서 작성한 문장들을 토대로 유저 1이 작성한 게시판의 글 내용을 토대로 댓글이 어떠한 것들이 있는지 알아보기 위해서는 아래와 같은 질의로 해결을 할 수가 있다.

```
post = db.bbs01.post.find( $and : { writeUser : "유저1", title : "게시판 글" })
commentList = db.bbs01.comment.find( _id : { $in : post.comment }).toArray()
```

이처럼 작성을 한 관계는 이미 짐작했듯이 Relational Database에서 흔히 쓰이고 있는 **One To Many**와 같은 개념으로 인지할 수 있다. 부모 Document와 자식 Document이 각각 자신만의 Document를 가지게 되어서 쉽게 관리를 할 수 있도록 도와주는 점에다가 Many To Many 관계에 대해서 각 Document에 작성만 해 주면 금방 해결해 주는 좋은 이득이 있지만 한 편으로 단점이 있다면 위처럼 좀 복잡한 Query 문을 작성을 하여 결과를 얻을 수 있기 때문에 이에 대해서는 아직까지 효율적이라고 생각을 할 수가 없다. 왜냐면 MongoDB 자체 내에서 단일 Document 크기가 16MB로 한정이 되어 있기 때문에 이렇게 많은 관계들에 대해서 자식 Document에 적용을 하는 것도 다시 생각을 해 봐야 할 때이다. 그래서 이번에는 방금 전과 달리 Comment에 부모 Document의 ObjectId를 작성을 해 본다면 아래와 같이 작성할 수 있다.

```
[{
   _id : "0001",
   writeUser : "유저1",
   title : "게시판 글",
   context : "게시판 내용",
}]
```

```
> db.bbs01.post.find()
```

```
[{ _id : "A001", writeUser : "유저2", context : "댓글1", post : ObjectId("0001") }, { _id : "A002", writeUser : "유저3", context : "댓글2", ObjectId("0001") }]
```

```
> db.bbs01.comment.find()
```

이는 부모 Document의 _id의 값을 자식 Document가 가지고 있어서 어떻게 본다면 자식 Document의 수가 더욱 많아지는 경우에 이를 이용하면 좋겠지만 이러한 방안을 **One To Squillions** 관계로 칭하겠다. One To Many를 이용할지 One To Squillions 관계를 이용할지에 대해서 많은 의문이 남아지고 있는데 이러한 면에서는 어떻게 본다면 Relational Database에서는 대부분 부모 테이블을 Eager 정책으로 데이터 로딩을 하였지만, MongoDB에서는 부모 Document든 자식 Document든 어느 것을 Eager 정책으로 선택을 할 수 있는 폭이 주어지는 점에서 생각을 해 본다면 실제로 데이터 모델링을 하는 경우에 객체의 관계를 더더욱 고려를 해야 할 필요도 있다는 점을 느낄 수 있다.

이를 토대로 방금 전과 같은 연산을 하게 된다면 다음 페이지에서 볼 수 있다. 그렇지만 Join과 같은 개념으로 질의를 작성하게 되니 아까 같이 복잡한 건 여전한 편이다.

```
post = db.bbs01.find({ _id : ObjectId("0001")})
commentList = db.bbs01.comment.find({ post : post._id })
```

C. 비정규화(조회를 많이 할 때)

Relational Database에서는 데이터의 중복을 줄여서 정규화를 강조하고 있지만, MongoDB에서는 데이터의 중복을 오히려 포함을 해서 비정규화를 통하여 데이터의 관계를 보장을 하고 있다. 위에서 방금 언급했던 One To Many, One To Squillions 관계에 대해서 비정규화를 이용해서 작성을 해 본다면 아래와 같이 볼 수 있다.

> One To Many 관계

```
[{
   _id : "0001",
   writeUser : "유저1",
   title : "게시판 글",
   context : "게시판 내용",
   comment : [ { id : ObjectId("A001"), writeUser : "유저2" }, { id : ObjectId("A002"), writeUser : "유저3" } ]
}]
```

> db.bbs01.post.find()

```
[{ _id : "A001", writeUser : "유저2", context : "댓글1" }, { _id : "A002", writeUser : "유저3", context : "댓글2" }]
```

> db.bbs01.comment.find()

이처럼 비정규화 작업을 하게 된다면 오히려 자식 Document들의 데이터를 중복으로 이용을 할지 언정 자식 Document들의 데이터를 무조건 데려 오는 것을 사전에 방지할 수 있는 점에 대하여 장점을 보일 수가 있다. 그렇지만 위에서는 설명을 못 하고 넘어갔는데 데이터를 갱신할 경우에는 comment의 Document에 포함된 이름도 변경을 해야 하는 단점도 더러 존재하기 때문에 갱신을 적게 하는 경우에 쓰기를 권장을 한다. 이를 이용한 쿼리문을 재작성한다면 아래와 같이 작성할 수 있다. 쿼리는 위에서 언급한 내용과 그대로이다.

```
post = db.bbs01.find($and : { _id : ObjectId("0001"), writeUser : "유저1" })
comment_ids = post.comment.map(
     function(com) {
         return com.id
     } )
commentList = db.bbs01.find({ _id : { $in : comment_ids } })
```

> One To Squillions 관계

```
[{
   _id : "0001",
   writeUser : "유저1",
   title : "게시판 글",
   context : "게시판 내용"
}]
```

> db.bbs01.post.find()

```
[{ _id : "A001", writeUser : "유저2", context : "댓글1", post : { id : Object("0001"), writeUser : "유저1" },
 { _id : "A002", writeUser : "유저3", context : "댓글2", post : { id : Object("0001"), writeUser : "유저1" }]
```

> db.bbs01.comment.find()

이도 마찬가지로 post에는 id와 writeUser를 남겼는데 언젠가 Post의 내용이 변경 되는 경우에는 이를 반영을 해줘야 하는 것도 마찬가지로 볼 수 있다. 쿼리 문장을 방금 전처럼 다뤘던 내용으로 작성을 한다면 아래와 같은 문장으로 작성을 할 수 있다.

```
post = db.bbs01.find({ _id : "0001"})
commentList = db.bbs01.find()
comment_ids = commentList.map(
                    function(com){
                        if(com.post.id === post._id) return com.post.id
                    })
commentList = db.bbs01.find({ _id : { $in : comment_ids } })
```

D. DBRef

Relational Database에서 One To Many, One To Squillious 관계에 대해서 적용을 할 때에는 ObjectId를 이용해서 작성을 한다면 크게 상관은 없지만 Spring Data MongoDB에서 One To Many 관계를 적용한 객체 관계 매핑 작업을 하게 된다면 현재 작성하는 프로젝트 내에서 관계성을 보장하는 기능도 추가를 해야 하기 때문에 생각보다 까다로워진다. 또한 필자도 공공데이터 Excel 파일에서 제공하는 데이터의 관계를 형성하기 위해서 이에 대하여 많은 고민을 했는데 다행이 이를 해결할 수 있는 DBRef 개념이 있다. 이에 대해서 꼭 인지를 하고 넘어가야 Spring Data MongoDB에서 어려움 없이 잘 해결할 수 있다. 그래서 이번에는 MongoDB에서 관계에 대한 데이터를 저장을 함으로서 Reference하는데 있어서 지장이 없도록 하는 방안을 공부를 하고 넘어가겠다.

```
필드_이름 : { $ref : <콜렉션 이름>, $id : <ObjectId>, $db : <타 데이터베이스 이름> }
```

이를 적용한 문장을 토대로 One To Squillious 관계에 적용을 한다면 아래와 같이 작성을 할 수 있다.

```
[{ _id : "A001", writeUser : "유저2", context : "댓글1", post : { $ref : "post", $id : ObjectId("0001"), $db : "bbs01" }
 { _id : "A002", writeUser : "유저3", context : "댓글2", post : { $ref : "post", $id : ObjectId("0001"), $db : "bbs01" }]
> db.bbs01.comment.find()
```

우리가 현재 공부를 해봤던 MongoDB Relationship을 정리를 한다면 아래와 같이 할 수가 있다. 다만 이에 대한 이야기는 모든 데이터베이스에서 해당이 되는지에 대해서 확인을 할 필요가 있으며 Many To Many 관계에 대해서도 더욱 효율적으로 도와주는 역할을 한다는 점도 알아 두면 좋겠다.

One To Few	One To Many	One To Squillions
Embedded Document	Reference Document	
자식의 검색 비중이 적은 데이터	부모의 검색 비중이 적은 데이터	자식의 검색 비중이 적은 데이터
1 : 1, 1 : N	1 : N, M : N	N : 1, M : N
Foreign ObjectId		
데이터 갱신(Update)에 대해서 어려움없이 가능하다. 쓰기 비중이 많을 때 좋음.		
비정규화		
데이터 갱신에 대해 어렵기 때문에 읽기 비중이 많을 때 좋음.		

3. MongoDB Database Modeling에서 고려해야 할 점

MongoDB 데이터 모델링의 6가지 원칙에 대해서 잠깐 알아보고 간다면 방금 전에 읽어 봤던 내용들이 생각보다 좀 복잡하고 어렵기 때문에 잘 이해가 가질 않은 사람들도 더러 있을 것이다. 이를 읽어보면서 추후에 NoSQL의 대표 주자인 MongoDB를 이용해서 모델링을 할 때에는 이들의 고려사항들을 인지해서 작성을 해 보도록 하자.

1. 피할 수 없는 이유가 없다면 Document에 포함할 것.

   - 이는 즉, One To Few로 작성을 해도 될 Document에 대한 이야기이다. 굳이 Collection으로 생성을 해서 만들 필요가 없다면 이에 대해 고려를 해 봐야 한다.

2. 객체에 직접 접근할 필요가 있다면 Document에 포함하지 않아야 한다.

   - 데이터들 중에서 열거를 할 수 있을 정도로 정리가 된 경우에는(즉 종류가 부지기수 하지 않고 대여섯 종류로 줄일 수 있을 때) Document에 포함시키는 것보다는 하나의 카테고리처럼 정리를 하는 것이 올바른 판단이라는 뜻이다.

3. 배열이 지나치게 커져서는 안 된다. 자식 Document의 데이터 크기가 적당하면 One To Many, 반면에 이 Document의 규모가 커지면 One To Squillions으로 조정을 해야 한다.

   - 이는 즉 부모 Document와 자식 Document의 선택을 제대로 하여 실제 모델링에서 16MB의 용량을 초과하지 말라는 뜻으로 생각하면 된다.

4. Join에 대해서는 염두를 하지 말고 넘어가자. Join은 Index를 잘 지정한다면 관계형 데이터베이스의 Join

과 비교를 하더라도 큰 변화가 없다.

5. Application에서의 Join을 할 수도 있는데 Join을 하는 비용이 각 분산된 데이터를 찾아 갱신하는 성능과 시간이 많이 든다면 비정규화를 고려해야 한다. 즉, 비정규화는 읽기, 쓰기의 비율에 따라서 설정을 해줘야 한다.

6. MongoDB 데이터 모델링에 대해서는 어플리케이션 데이터 접근 패턴에 달려 있어서 어떻게 읽고 어떻게 갱신을 하는가에 대한 여부를 따라서 달라진다.

   - MongoDB의 데이터 모델링도 중요하지만 무엇보다 가장 중요한 것은 Web Application에서 어떻게 구성을 하였는가에 대해서도 2차로 따지고 생각해 볼 수 있기 때문에 이에 대해서도 게을리 생각하지 말라는 뜻이다.

그리고 위에서 언급한 사항은 실제로 고려를 해야 하는 점들이지만 일반적인 규칙은 아래와 같이 적용을 하고 있다.

- 클래스 객체가 최상위 레벨이면 자신만의 Collection을 가진다.
- 아이템 세부 사항들은 Embedded Document로 쓴다.
- Embedded 관계로 모델링 된 객체들은 내장을 한다.
- Many To Many 관계는 참조를 하는 경우가 많다.
- 객체 목록이 그렇게 많지 않은 경우에는 전체 Collection을 빠르게 캐쉬 처리를 할 수 있도록 분리를 한다.
- 내장 객체로의 DBRef는 가질 수가 없게 된다. 실제로 내장 객체는 Collection에서 최상위 레벨인 객체보다 참조하기 어려워진다.
- 내장 객체 시스템 레벨의 뷰를 얻는 것도 어렵게 느껴질 것이다. 또한 내장하는 Document의 양이 커진다면 단일 사이즈 제한에 걸리게 된다.
- 성능 이슈가 존재한다면 내장을 하는 편이 좋다.

[출처]

http://victorydntmd.tistory.com/30 / http://tech.devgear.co.kr/db_kb/331

RDBMS에서의 관계형 설명

http://cyberx.tistory.com/166

NoSQL VS RDBMS 관계의 차이점

https://www.haruair.com/blog/3055 / https://blog.outsider.ne.kr/655

MongoDB Relationship 종류와 활용 방법