8. 인덱싱과 쿼리 최적화

YBIGTA Engineering Team 14기 이용하

INDEX

1.1 개념실험

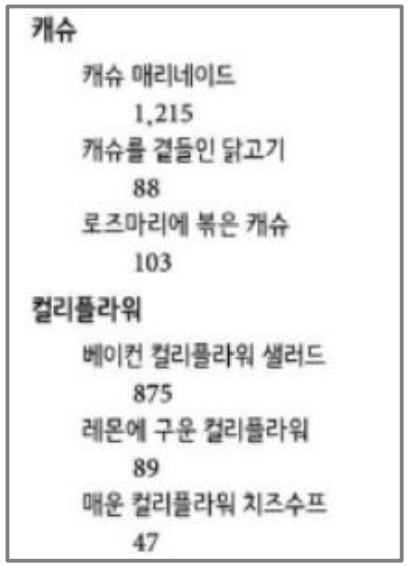
- 단순 인덱스 : **단일 키**

ex 1) 이름에 대한 인덱스

ex 2) 재료에 대한 인덱스

- 복합 인덱스 : **하나 이상의 키**

ex) 재료, 이름에 대한 인덱스



1.1 개념실험

인덱싱 규칙

1. **인덱스는 도큐먼트를 가져오기 위해 필요한 작업량을 많이 줄인다.** 적당한 인덱스가 없으면 질의 조건을 만족할 때까지 모든 도큐먼트를 차례로 스캔해야만 하고, 이것은 종종 컬렉션 전체를 스캔하는 것을 뜻한다.

2. **한 쿼리를 실행하기 위해서 하나의 단일 키 인덱스만 사용할 수 있다. 복합 키를 사용하는 쿼리에 대해서는 복합 인덱스가 적당하다.**

3. 재료-이름에 대한 인덱스를 가지고 있다면 재료에 대한 인덱스는 없앨 수 있고 또 없애야만 한다. 좀 더 일반적으로 표현해서 **a-b에 대한 복합 인덱스를 가지고 있다면 a에 대한 인덱스는 중복이고, b에 대한 인덱스는 중복이 아니다.**

4. **복합 인덱스에서 키의 순서는 매우 중요하다.**

1.2 인덱싱 핵심 개념

단일 키 인덱스 인덱스 내의 각 엔트리는 인덱스되는 도큐먼트 내의 한 값과 일치

ex) _id에 대해 디폴트로 생성되는 인덱스

복합 키 인덱스 복합 키 인덱스 : 키의 순서가 중요함. 정확히 일치하는 값 - 범위 지정

ex) 가격이 75달러보다 싸고 제조사가 'Acme'인 모든 상품을 찾는 query

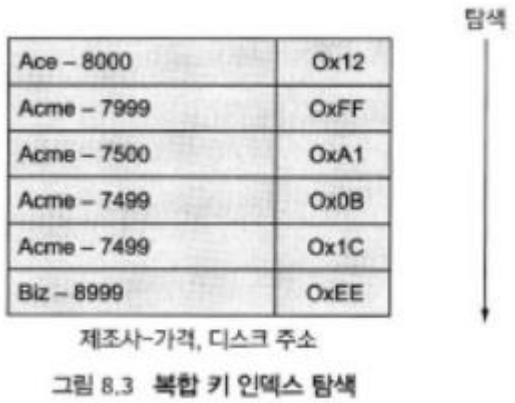


그림 8.3 복합 키 인덱스 탐색

그림 8.4 키의 순서가 바뀐 복합 키 인덱스

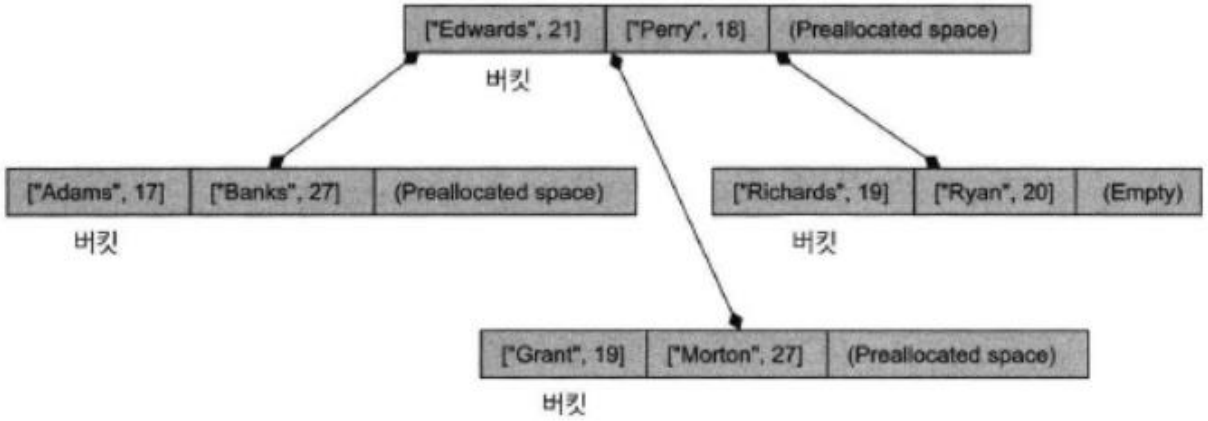
1.2 B-트리

MongoDB 내부적으로 **B-트리(B-tree)**로 인덱싱을 생성

1) 다양한 쿼리를 용이하게 처리

ex) 정확한 일치, 범위 조건, 정렬, 프리픽스 일치, 인덱스만의 쿼리 등

2) 키가 추가/삭제되어도 밸런스된 상태 유지



2.1 인덱스 타입

고유 인덱스 도큐먼트의 필드가 해당 도큐먼트에서 고유하도록 하는 특성

- unique 옵션 지정하여 고유 인덱스 생성

```
db.users.createIndex({username: 1}, {unique: true})
```

- 중복 데이터의 도큐먼트 삽입 시 실패

```
E11000 duplicate key error index:
  gardening.users.$username_1 dup key: { : "kbanker" }
```

컬렉션에 데이터가 존재하지 않을 때
고유 인덱스 생성하는 것이 좋음

2.1 인덱스 타입

- 기존에 존재하는 컬렉션에 대해 고유 인덱스 생성할 때

1) 고유 인덱스 생성 반복적으로 수행하며, 발생하는 실패 메시지를 이용해 중복 키 제거

2) dropDups 옵션으로 중복 키 가지는 도큐먼트 자동 삭제

```
db.users.createIndex({username: 1}, {unique: true, dropDups: true})
```

2.1 인덱스 타입

밀집 인덱스 컬렉션 내의 한 도큐먼트가 인덱스 키를 가지고 있지 않아도 인덱스에는 해당 엔트리가 존재

희소 인덱스 인덱스의 키가 null이 아닌 값을 가지고 있는 도큐먼트만 존재

1) 밀집 인덱스가 바람직하지 않을 때

: 모든 도큐먼트가 다 가지고 있지 않은 필드에 대해 고유 인덱스 생성하는 경우

```
db.products.createIndex({sku: 1}, {unique: true, sparse: true})
```

2) 컬렉션에서 많은 수의 도큐먼트가 인덱스 키를 가지고 있지 않은 경우

2.1 인덱스 타입

ex 1) 도큐먼트의 sku 필드에 대해 고유 인덱스 생성

→ sku에 대한 값 없이 상품을 시스템에 입력할 수 있을 때

➡ **sku 필드 값이 없는 도큐먼트를 여러 개 삽입하려고 하면 첫 번째를 제외한 나머지의 삽입 연산 실패** (인덱스에 sku가 null인 entry가 이미 존재)

ex 2) 상품명을 익명으로 할 수 있는 전자상거래 사이트

→ 리뷰의 반이 user_id 필드를 가지고 있지 않을 때

➡ **희소 인덱스를 이용해 user_id 필드를 통해 사용자와 연결되어 있는 상품평만을 인덱스로 설정 가능**

2.1 인덱스 타입

다중키 인덱스 필드의 값이 배열인 경우, 인덱스 내의 여러 개의 엔트리가 동일한 도큐먼트를 지시

ex) 여러 개의 태그를 갖는 상품 도큐먼트 가정

```
{
    name: "Wheelbarrow",
    tags: ["tools", "gardening", "soil"]
}
```

tags 필드에 대해 인덱스를 생성하면, 태그 배열에 있는 각 값들이 인덱스에 나타남

➡ 태그 배열의 값 중 어느 것으로도 도큐먼트를 찾을 수 있음

2.1 인덱스 타입

해시 인덱스 엔트리가 해시 함수를 통해 처음 전달. 해시된 값이 순서를 결정

```
db.recipes.createIndex({recipe_name: 'hashed'})
```

1) B Tree가아닌 Hash 자료구조를 사용
   - Hash는 검색 효율이 B Tree보다 좋지만, 정렬을 하지 않는다
2) 동등 쿼리는 거의 동일하게 작동하지만, 범위 쿼리는 지원되지 않는다
3) 다중 키 해시 인덱스는 허용되지 않는다
4) 부동 소수점 값이 해시되기 전에 정수로 반환된다

2.1 인덱스 타입

지리공간적 인덱스

각 도큐먼트에 저장된 위도값과 경도값에 따라 도큐먼트를 **특정 위치에 가까이 배치**

ex) 레스토랑 디렉터리 → 집 근처에 가장 가까운 레스토랑?

➡ 반경 5킬로미터 이내에 있는 모든 레스토랑을 찾기 위해 쿼리 실행

2.2 인덱스 관리

인덱스 생성과 삭제

- 생성 : createIndex()

```
use green
db.users.createIndex({zip: 1})
```

- 삭제 : deleteIndexes()

```
use green
db.runCommand({deleteIndexes: "users", index: "zip"})
```

dropIndex()

```
use green
db.users.dropIndex("zip_1")
```

- 인덱스에 대한 사항 확인 : getIndexes()

2.2 인덱스 관리

인덱스 구축

1. 생성할 인덱스 선언 `db.values.createIndex({open: 1, close: 1})`

2. 인덱스할 값 정렬. 정렬된 데이터는 효율적으로 B-트리에 추가됨

3. 정렬된 값들이 인덱스로 삽입

4. 인덱스 생성의 진척 상황 확인
   - MongoDB 서버 로그 확인
   - currentOP()

2.2 인덱스 관리

- 어떤 클라이언트도 인덱스가 생성되는 동안에는 데이터베이스에 읽거나 쓰기를 할 수 없음
- 데이터베이스가 실제 서비스 환경에 놓여 있다면 시간이 많이 소요되는 인덱스 구축은 바람직하지 않음

백그라운드 인덱싱　　**오프라인 인덱싱**

2.2 인덱스 관리

백그라운드 인덱싱

데이터베이스가 실제 서비스되고 있고, 데이터베이스에 대한 엑세스를 중지시킬 수 없을 때

➡ 인덱스가 백그라운드에서 구축되도록 지정

- 쓰기 잠금 가능하지만, 데이터베이스에 대한 다른 읽기나 쓰기를 허용하기 위해 잠시 멈춤

```
db.values.createIndex({open: 1, close: 1}, {background: true})
```

2.2 인덱스 관리

오프라인 인덱싱

- 백그라운드에서 인덱스를 생성하면 상용 서버에 허용할 수 없는 많은 양의 로드를 발생시킬 때

1. 한 복제 노드를 오프라인 상태로 바꿈
2. 그 노드에 대해 인덱스를 구축하고, 마스터 노드로부터 업데이트를 받음
3. 업데이트 완료 후 이 노드를 프라이머리 노드로 변경
4. 다른 세컨더리 노드들을 오프라인 상태로 바꾸고, 각 노드에 대해 인덱스를 구축함

2.2 인덱스 관리

Defragmenting

애플리케이션에서 기존 데이터의 업데이트나
대량의 데이터 삭제가 자주 발생하면 인덱스가 심하게 단편화

- 단편화된 인덱스 : 주어진 데이터의 크기 < 인덱스의 크기 → 인덱스가 램을 필요 이상으로 사용

➡ 인덱스 재구축 : 개별 인덱스를 삭제하고 재생성

```
db.values.reIndex();
```

reIndex() : 해당 컬렉션에 있는 모든 인덱스 재구축

3.1 느린 쿼리 탐지

대부분의 애플리케이션에서 쿼리가 100밀리초 이내에 실행되어야 함

➡ MongoDB 로거 : 어떠한 연산이라도 100밀리초를 넘어서면 경고 메시지 프린트

- 경고 메시지 확인 : grep -E '[0-9]+ms' mongod.log
- 임계값 조정 : --slowms (임계값)

3.1 느린 쿼리 탐지

프로파일러 사용

1. 프로파일링하려는 데이터베이스 선택
2. 프로파일링 수준 지정

```
use stocks
db.setProfilingLevel(2)
```

→ 0 : 사용 불가능 상태 / 1 : 느린 연산만 로그로 남김 / 2 : 최대 출력

```
use stocks
db.setProfilingLevel(1, 50)
```

→ 어떤 임계값을 넘어서는 연산만을 로그로 기록하는 옵션 지정

3. 쿼리 실행

```
db.values.find({}).sort({close: -1}).limit(1)
```

→ 주식 데이터에서 최고 종가 찾기

3.1 느린 쿼리 탐지

프로파일링 결과

- 결과는 프로파일러를 실행한 데이터베이스에 있는 system.profile 캡드 컬렉션에 저장
- system.profile에 질의 가능

```
db.system.profile.find({millis: {$gt: 150}})
```

→ 150 밀리초 이상이 소요된 모든 쿼리

- 캡드 컬렉션은 삽입된 순서 유지

```
db.system.profile.find().sort({$natural: -1}).limit(5).pretty()
```

→ $natural 오퍼레이터를 사용해서 가장 최근의 결과 먼저 출력

3.1 느린 쿼리 탐지

프로파일링 전략

1. 적당히 높은 값으로 시작해서 점점 값을 줄여나간다
2. 프로파일러가 사용 가능 상태로 있는 동안에도 애플리케이션이 원래대로 작동해야 한다
   - 애플리케이션의 모든 읽기 및 쓰기를 테스트해야 함
3. 애플리케이션이 제대로 작동하기 위해 데이터의 크기와 쿼리 로드, 하드웨어가 애플리케이션의 실제 서비스 환경과 동일한 조건하에서 읽기와 쓰기가 수행되어야 한다

3.2 느린 쿼리 분석

EXPLAIN() 주어진 쿼리의 경로에 대한 자세한 정보 제공

1.

```
db.values.find({}).sort({close: -1}).limit(1).explain()
{
        ...
        “n” : 1,
        “nscanned” : 4308303,
        ...
}
```

→ n : 반환된 수, nscanned : 스캔된 수

```
db.values.count()
4308303
```

→ 스캔한 도큐먼트의 수가 컬렉션 내의 도큐먼트 수와 같다. 즉, **컬렉션 자체를 스캔한 것**

➡ **N과 nscanned가 비슷한 값을 가져야 함**

3.2 느린 쿼리 분석

2. scanAndOrder 필드

- 쿼리 옵티마이저가 인덱스를 사용하지 못하고 정렬된 결과값을 반환할 때 나타남

➡ **쿼리 프로세서가 '컬렉션 스캔 + 결과값 직접 정렬' 해야 함**

느린 쿼리 해결책

1) 인덱스 생성
2) 인덱싱된 키 사용

3.2 느린 쿼리 분석

쿼리 옵티마이저

규칙

1) scanAndOrder를 피한다. 즉, **쿼리가 정렬을 포함하고 있으면 인덱스를 사용한 정렬을 시도**한다.
2) 유용한 인덱스 제한 조건으로 모든 필드를 만족시킨다. 즉, **쿼리 셀렉터에 지정된 필드에 대한 인덱스를 사용**하도록 노력한다.
3) 쿼리가 범위를 내포하거나 정렬을 포함한다면 **마지막 키에 대한 범위나 정렬에 도움이 되는 인덱스를 선택**하라.

3.2 느린 쿼리 분석

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}})
```

→ 구글에 대해 200이 넘는 모든 종가를 찾는 쿼리

```
db.values.createIndex({stock_symbol: 1, close: 1})
```

→ 쿼리에 대한 최적의 인덱스는 두 개의 키를 모두 포함하고 범위 쿼리를 고려해서 close 키가 마지막으로 와야 함

```
db.values.find({stock_symbol: “GOOG”, close: {$gt: 200}}).explain()
{
        …
        “n” : 730,
        “nscanned” : 730,
        …
}
```

n 값 = nscanned 값 : **쿼리가 최적이다**

3.3 쿼리 패턴

단일 키 인덱스

- 정확한 일치

```
db.values.find({close: 100})
```

- 정렬

```
db.values.find({}).sort({close: 1})
```

- 범위 쿼리

```
db.values.find({close: {$gte: 100}})
```

3.3 쿼리 패턴

복합 키 인덱스

쿼리당 하나의 범위나 정렬을 할 때만 효율적

- 정확한 일치

```
db.values.find({close: 1})
db.values.find({close: 1, open: 1})
db.values.find({close: 1, open: 1, date: "1985-01-08"})
```

- 범위 일치

```
db.values.find({}).sort({close: 1})
db.values.find({close: {$gt: 1}})
db.values.find({close: 100}).sort({open: 1})
db.values.find({close: 100, open: {$gt: 1}})
db.values.find({close: 1, open: 1.01, date: {$gt: "2005-01-01"}})
db.values.find({close: 1, open: 1.01}).sort({date: 1})
```

- 커버링 인덱스

  특별히 질의가 필요로 하는 모든 데이터가 인덱스 내에 있을 경우

```
db.values.find({close: 1}, {open: 1, close: 1, date: 1, _id: 0})
```

4 요약

- 인덱스는 매우 유용하지만 많은 비용을 수반하므로 쓰기가 느려진다
- 일반적으로 MongoDB는 쿼리에서 하나의 인덱스만 사용하므로 여러 필드의 쿼리는 복합 인덱스가 효율적일 것을 요구한다
- 복합 입덱스를 선언할 때 순서가 중요하다
- 비용이 많이 드는 쿼리를 계획하되 피하는 것이 좋다. MongoDB의 explain 명령, 비용이 많이 드는 쿼리 로그 및 프로파일러를 사용하여 최적화된 쿼리를 찾는다
- MongoDB는 인덱스를 작성하는 몇 가지 명령을 제공하지만, 항상 비용이 들어 애플리케이션을 방해할 수 있다. 이는 트래픽이나 데이터가 많아지기 전에 쿼리를 최적화하고 인덱스를 초기에 만들어야 함을 뜻한다
- 스캔한 도큐먼트 수를 줄임으로써 쿼리를 최적화한다. explain 명령은 쿼리가 하는 일을 발견하는 데 매우 유용하다. 최적화를 위한 지침으로써 이를 사용하도록 하자

Thank You !