인덱스 개요

MongoDB의 인덱스 기본

인덱스 아키텍처, 동작 원리

- wired-tiger
  - 인덱스는 b-tree 구조 이지만 record-id 는 엔진내부에서 cluter index 형태로 구성되어 있다.
  - 이때문에 mmapv4 보다 조회는 느리지만 변경에 대한 부하는 적다.
- 로컬 인덱스
  - mongodb 는 샤드 단위의 유니크함만 포함하므로 클러스터 입장에서 본다면 로컬 인덱스 이다.
  - secondary 인덱스 전부 (primary key , unique key 는 샤드키를 포함했다는 전제)
  - shard key 를 index 에 포함하지 않으면 unique 함을 보장하지 않음
- 인덱스 키 엔트리 자료구조
  - collection 과 달리 index 는 스키마 프리가 아니다.

MongoDB의 다양한 인덱스

B-tree 인덱스

collection 과 달리 index 는 컬럼의명세를 가진다. 서브 도큐먼트를 index 로 했을때의 주의점

- 단일값과 달리 서버도큐먼트 필드의 키-값 전부를 값으로 간주하여 함께 저장한다.
  - 서브다큐먼트의 필드명 길이가 중요함
- 조건절 에서는 서브도큐먼트의 필드를 모두 가지고 있어야 하며 순서도 같아야 한다.
  - 실제 테스트 했을때는 조회에서 조건절에서 모든 필드가 가져야 하거나 순서가 바뀌어도 상관 없음

테스트

```
db.users.find();
{ "_id" : ObjectId("601a1e98fb91390a854cf027"), "name" : { "first" : "john",
"last" : "kennedy" } }
{ "_id" : ObjectId("601a3c13fb91390a854cf028"), "name" : { "first" : "john2",
"last" : "kennedy2" } }
{ "_id" : ObjectId("601a3c1afb91390a854cf029"), "name" : { "first" : "john3",
"last" : "kennedy3" } }

> db.users.getIndexes();
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_"
        },
```

```
        {
                "v" : 2,
                "key" : {
                        "name" : 1
                },
                "name" : "name_1"
        }
]
>

// dot notation 이용시
// winning plan , "stage" : "COLLSCAN",
db.users.find( { "name.first": "john" , "name.last": "kennedy" } ).explain();
db.users.find( { "name.first": "john" } );

// winning plan , "stage" : "IXSCAN",
db.users.find( { name: { last:"kennedy",first:"john"} } ).explain();
db.users.find( { name: { first:"john", last:"kennedy"} } ).explain();
db.users.find( { name: { first:"john" } } ).explain();
db.users.find( { name: { last:"kennedy"} } ).explain();
```

프리픽스 스캔

문자열을 대상으로 일부만 일치하는 패턴을 검색할 때 사용할 수 있다.

```
db.users.find( { name: {$regex: /^chlee/ } } )
db.users.find( { name: /^chlee/ } )
```

프리픽스 스캔은 SQL 의 LIKE "chlee%" 와 같은 연산이며 다음 조건을 만족해야 한다.

- "^" 로 시작하기
- "^" 이외에 정규 표현식 포함하지 않기

커버링 인덱스

커버링 인덱스는 쿼리의 결과가 인덱스 필드인 경우 collection 을 억세스 하지 않고 인덱스에서 바로 결과값을 반환하는 경우를 말한다.

인덱스 인터섹션

쿼리를 수행할때 복수의 인덱스를 이용하여 각각의 결과를 도출하고 교집합하여 결과를 최적화 하는 방식을 말한다.

- 사용자가 지정하여 사용할 수 없다.
- 두개 이상의 인덱스가 사용될 수 있다.
- explain() 에서 사용여부를 확인할 수 있다.
    - AND_SORTED or AND_HASH 스테이지

Hash 인덱스

특징

- 단일 건에 대해서는 특화되어 있다.
  - md5 값, 8바이트로만 인덱스가 구성된다.
  - field 가 평균 8바이트 이상이라면
  - 메모리, 디스크 i/o 에서 성능상 유리하다.
- 부동소수점은 64bit 값으로 변환한다.(truncate)
- hash 값 확인하기
  - convertShardKeyToHashed() 를 통해 hash 값 확인 가능
- hash 인덱스 구성은 b-tree
  - 키 는 hashed 값을 가지나 구성은 b-tree 로 되어 있다.

제약사항

- array 필드를 지원하지 않는다.
- hash 값으로 인덱스를 구성하기 때문에 범위 검색에 사용할 수 없다.
- hash 값으로 인덱스를 구성하기 때문에 정렬을 보장하지 않음
- compound index 는 single filed hashed 만 생성할 수 있다.
  - 인베디드 다큐먼트를 이용하여 compound index 와 유사하게 만들 수 있다.
  - 단 검색시 필드의 순서가 맞아야 한다.

hash인덱스 만들기

```
// single field hash index
db.collection.createIndex( { _id: "hashed" } )

// compound hash index with single hashed field
db.collection.createIndex( { "fieldA" : 1, "fieldB" : "hashed", "fieldC" : -1 } )
```

> hash 인덱스는 키의 평균길이가 8바이트 보다 적으면 동등 비교시에만 유리하다.

멀티키 인덱스

멀티키 인덱스는 배열같이 키값이 복수로 구성된 키를 대상으로 하는 인덱스 이다.

제약사항

- compound 인덱스 생성시 배열필드는 하나만 가능하다.

```
// { a: 1, b: 1 }  처럼 복수의 배열 필드를 대상으로 compound 인덱스를 생성할 수 없다.
{ _id: 1, a: [ 1, 2 ], b: [ 1, 2 ], category: "AB - both arrays" }

// { a: 1, b: 1 } 로 compound 인덱스 생성시 두 번째 데이터는 입력할 수 없다.
{ _id: 1, a: [1, 2], b: 1, category: "A array" }
{ _id: 2, a: 1, b: [1, 2], category: "B array" }

// { "a.x": 1, "a.z": 1 } 로 생성시 아래 문서는
// 모두 배열 필드는 하나인것을 만족하기 때문에 가능하다.
```

```
{ _id: 1, a: [ { x: 5, z: [ 1, 2 ] }, { z: [ 1, 2 ] } ] }
{ _id: 2, a: [ { x: 5 }, { z: 4 } ] }
```

- 인덱스 엔트리가 늘어날 수록 INSERT 의 성능 저하가 클 수 있다.
- 멀티키 인덱스는 샤드키로 사용될 수 없다.
- 해시 알고리즘을 사용하는 인덱스는 멀티키 인덱스로 정의될 수 없다.
- 멀티키 인덱스는 커벙링 인덱스 처리가 안된다.
-

멀티키 인덱스 bound

멀티키 인덱스의 스캔범위는 일반 인덱스와 다르다.

```
db.survey.insert({ _id:1, item: "ABC", ratings: [2,9] })
db.survey.insert({ _id:2, item: "XYZ", ratings: [4,3] })
db.survey.createIndex({ratings:1})

// 일반 인덱스라면 raings :[4,3] 만 나와야 한다.
> db.survey.find({ratings: { $gte:3, $lte:6 }})
{ "_id" : 1, "item" : "ABC", "ratings" : [ 2, 9 ] }
{ "_id" : 2, "item" : "XYZ", "ratings" : [ 4, 3 ] }
```

멀티키 인덱스는 위 쿼리에 대해 다음과 같이 처리한다.

```
// 두 결과의 union
ratings: {$gte: 3}
ratings: {$lte: 6}
```

이런 경우 $elemMatch 를 사용할 수 있다.

```
> db.survey.find({ratings: {$elemMatch:{$gte:3, $lte:6}}})
{ "_id" : 2, "item" : "XYZ", "ratings" : [ 4, 3 ] }

// $elemMatch 를 사용하면 두 결과의 교집합으로 으로 반환한다.
ratings: {$gte: 3}
ratings: {$lte: 6}

// 주의 할 점은 배열 요소 중 하나라도 일치한다면 true 로 간주 한다.
db.survey.insert({ _id:3, item: "MNQ", ratings: [2,4] })

> db.survey.find({ratings: {$elemMatch:{$gte:3, $lte:6}}})
{ "_id" : 2, "item" : "XYZ", "ratings" : [ 4, 3 ] }
{ "_id" : 3, "item" : "MNQ", "ratings" : [ 2, 4 ] }
```

서브 도큐먼트의 배열에 대해 멀티키를 사용했을 경우 주의 점

```
db.survey.insert( { _id:1, items: [ {item: "ABC1", rating:2}, {item: "ABC2",
rating:9} ] })
db.survey.insert( { _id:2, items: [ {item: "XYZ1", rating:4}, {item: "XYZ2",
rating:3} ] })
db.survey.insert( { _id:3, items: [ {item: "MNQ1", rating:2}, {item: "MNQ2",
rating:4} ] })
db.survey.createIndex( {"items.rating": 1})

// 잘못된 사용법
> db.survey.find( { "items.rating" : { $elemMatch: { $gte: 3, $lte: 6 } } } )
(결과없음)

> db.survey.find( { items :{ $elemMatch: { rating: {$gte: 3, $lte: 6 } } } } )
{ "_id" : 2, "items" : [ { "item" : "XYZ1", "rating" : 4 }, { "item" : "XYZ2",
"rating" : 3 } ] }
{ "_id" : 3, "items" : [ { "item" : "MNQ1", "rating" : 2 }, { "item" : "MNQ2",
"rating" : 4 } ] }
```

> 멀티키 인덱스 사용시 동등비교는 일반인덱스와 같지만 범위 스캔일 경우 bound 에 대해 주의 해서 사용해야 한다.

전문 검색 인덱스

특징

- 하나의 collection 당 하나의 인덱스만 생성할 수 있다.
- 쿼리에 $text 가 포함되는 경우 hint()를 사용할 수 없다.
- sort 작업은 전문검색 필드로는 작업을 할 수 없다. (b-tree 필드를 사용)

형태소 분석 vs n-gram

- 형태소 분석
- 문장에서 불용어(stop word) 를 제외하고 어근을 유니크하게 추출하는 방식
- mongodb 에서는 일정 언어에 대해서 형태소 분석만 지원한다.
- n-gram
- 모든 문자열을 일정은 길이로 잘라서 전부 가져가는 방식이다.

	형태소 분석	**n-gram**
언어 의존도	- 국가별로 알고리즘이 달라야 한다. - 개별 알고리즘으로 복잡하다. - 모든 단어의 어근을 찾기 어렵다.	- 언어의 특성을 고려하지 않는다. - 어근을 찾기위한 알고리즘도 필요 없다.
인덱스 크기	- 불용어를 필터링한 후 중복된 어근만 가지기 때문에 용량이 작다.	- 모든 문자열을 일정 크기로 전부 가진다. - 2개의 인덱스를 가진기 때문에 용량이 크다.

	형태소 분석	n-gram
성능	- 인덱스 용량이 작아 insert/delete 성능이 비교적 빠르다.	- 용량이 크므로 상대적으로 느리다.
검색 품질	- 알고리즘에 따라 품질 차이가 심하다 - 알고리즘 개선에 많은 비용이 든다.	- 적은 비용으로 평균적 품질을 보장할 수 있다.

전문 검색 인덱스 사용

```
// 필드 레벨
db.articles.createIndex( {title: "text"}, {name: "articles_name"})
db.articles.createIndex( {title: "text", contexts: "text"})

// 다큐먼트 레벨
db.articles.createIndex( {"$**": "text"})

// 삭제시 인덱스 이름으로만 삭제가 가능하다.
db.articles.dropIndex("articles_name")
```

중요도(weight) 할당

필드별로 중요도를 설정할 수 있으며, 데이터를 조회할 때 중요도를 기준으로 정렬해서 결과를 가져올 수 있다. 기본 중요도는 1이다.

결과에서 $meta 를 통해 "textScore" 를 반환하게 되는데 계산 방식은

- 단일 키워드로 동등할 경우 weight 값이 반환된다.
- 일부 매칭에는 0.75 * N (일치하는 단어 수) 로 계산된다.

```
db.scripts.insert({movie_name: "starwars", phrase: "droids"})
db.scripts.insert({movie_name: "starwars", phrase:"These are not the droids you
are looking for"})

db.scripts.createIndex( {movie_name:"text", phrase:"text"})

> db.scripts.find({$text:{$search:"droids looking"}}, {score:{$meta:"textScore"}})
{ "_id" : ObjectId("6058385a5a6ee739dcdb4bbd"), "movie_name" : "starwars",
"phrase" : "droids", "score" : 1 }
{ "_id" : ObjectId("6058385f5a6ee739dcdb4bbe"), "movie_name" : "starwars",
"phrase" : "These are not the droids you are looking for", "score" : 1.5 }
```

compound 인덱스

전문 검색 인덱스는 compound 다음과 같은 형태만 가능하다.

- b-tree 인덱스 + text 필드
    - b-tree 필드가 선행되었다면 검색시 Left-most match 규칙에 따라 해당 필드도 선행되어야 한다.

    - 그냥 인덱스 못탄다고 하면 안되나.. 정책에 일관성이 없어 보임
- text 필드 복수
    - text 필드는 들은 서로 인접해야 한다.

```
// b-tree 와 함께 생성되는 경우
// 인덱스 생성시 b-tree 의 필드가 선행되는 경우 검색조건에도 해당 필드가 함께해야 한다.
// 반대로 후위에 생성된다면 무관하다.
db.scripts.insert({jenre:"a", movie_name: "starwars", phrase: "droids"})
db.scripts.insert({jenre:"b", movie_name: "starwars", phrase:"These are not the
droids you are looking for"})

// 선생으로 생성시 실패
db.scripts.createIndex( {jenre:1, movie_name:"text", phrase:"text"})
> db.scripts.find({$text:{$search:"droids looking"}}, {score:{$meta:"textScore"}})
Error: error: {
        "ok" : 0,
        "errmsg" : "error processing query: ns=myDB.scriptsTree: TEXT :
query=droids looking, language=english, caseSensitive=0, diacriticSensitive=0,
tag=NULL\nSort: {}\nProj: { score: { $meta: \"textScore\" } }\n planner returned
error :: caused by :: failed to use text index to satisfy $text query (if text
index is compound, are equality predicates given for all prefix fields?)",
        "code" : 291,
        "codeName" : "NoQueryExecutionPlans"
}

// 후위로 생성시 성공
> db.scripts.createIndex( { movie_name:"text", phrase:"text" ,jenre:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
> db.scripts.find({$text:{$search:"droids looking"}}, {score:{$meta:"textScore"}})
{ "_id" : ObjectId("60583b405a6ee739dcdb4bc6"), "jenre" : "b", "movie_name" :
"starwars", "phrase" : "These are not the droids you are looking for", "score" :
1.5 }
{ "_id" : ObjectId("60583b405a6ee739dcdb4bc5"), "jenre" : "a", "movie_name" :
"starwars", "phrase" : "droids", "score" : 1 }

// text 필드들은 서로 인접해야 한다.
> db.scripts.createIndex( { movie_name:"text",jenre:1, phrase:"text" })
{
        "ok" : 0,
        "errmsg" : "Error in specification { key: { movie_name: \"text\", jenre:
1.0, phrase: \"text\" }, name: \"movie_name_text_jenre_1_phrase_text\", v: 2 } ::
caused by :: 'text' fields in index must all be adjacent",
        "code" : 67,
        "codeName" : "CannotCreateIndex"
}
```

파티셔닝

전문 검색 인덱스를 생성하고 분석된 어근의 데이터가 많은 경우 많은 인덱스 페이지를 검색해야 하기 때문에 다소 성능이 떨어 질 수 있다. 이때 b-tree 와 선행으로 인덱스를 생성할 경우 파티셔닝 효과를 볼 수 있다.

의문점, 검색범위가 줄어들어 파티션 테이블의 푸르닝 같은 효과를 이야기 하지만 구간 마다 중복 데이터가 있다면 어근의 데이터도 중복되는것 인지 궁금

예를 들어 다음과 같은 어근(a b c)이 있다고 가정했을때

```
{date:"2019-01" , contents: "a b c"}
{date:"2020-01" , contents: "a b c"}
```

파티셔닝을 하지 않으면 인덱스에는 "a b c" 를 하나씩만 가지고 있게 되지만 위 처럼 b-tree 형태로 구성할 경우 "a b c" 를 복수로 가지게 되므로 파티셔닝은 트레이드 오프 같다는 생각

언어 구분 및 대소문자 처리

언어 구분

```
{ _id: 1, idioma: "portuguese", quote: "A sorte protege os audazes" }
{ _id: 2, idioma: "spanish", quote: "Nada hay más surrealista que la realidad." }
{ _id: 3, idioma: "english", quote: "is this a dagger which I see before me" }

db.quotes.createIndex( { quote : "text" },
                       { language_override: "idioma" } )
```

대소문자 처리

```
> db.quotes.find( {$text:{$search:"Dagger", $language:"english",$caseSensitive:
true}})
(not found)
> db.quotes.find( {$text:{$search:"Dagger", $language:"english",$caseSensitive:
false}})
{ "_id" : 3, "idioma" : "english", "quote" : "is this a dagger which I see before
me" }
>
```

부정 비교와 문장 검색

하이픈("-")을 사용해 부정비교를 할 수 있다.

```
> db.quotes.find( {$text:{$search:"dagger see"}})
{ "_id" : 3, "idioma" : "english", "quote" : "is this a dagger which I see before
me" }
```

```
> db.quotes.find( {$text:{$search:"dagger -see"}})
>
```

문장 검색시 공백은 or 조건으로 인식 하고 역슬래쉬 더블쿼터는 and 로 인식 한다.

```
> db.quotes.find( {$text:{$search:"dagger chlee"}})
{ "_id" : 3, "idioma" : "english", "quote" : "is this a dagger which I see before
me" }
> db.quotes.find( {$text:{$search:" \"dagger\" \"chlee\" "}})
>
```

공간 검색 인덱스

인덱스의 속성

프라이머리 키와 세컨더리 인덱스

- 프라이머리 키
  - _id 라는 필드만 프라이머리 키가 될 수 있다.
  - 프라이머리 키가 없는 collection 은 생성할 수 없다.
  - hash 인덱스로 생성될 수 없다
  - compound 인덱스로 생성할 수 없다.
    - 서브 도큐먼트를 사용할 수는 있다.
    - "{ _id : {name: "chlee", address: "seoul"} }"
- 세컨더리 인덱스
  - 프라이머리 키를 제외한 모든 인덱스

유니크 인덱스

collection 을 대상으로 uniqueness 한 특성을 강제해서 인덱스를 생성할 수 있다.

```
// single field index
db.members.createIndex( { "user_id": 1 }, { unique: true } )
// compound index
db.members.createIndex( { groupNumber: 1, lastname: 1, firstname: 1 }, { unique:
true } )

// multikey index
{ _id: 1, a: [ { loc: "A", qty: 5 }, { qty: 10 } ] }
db.collection.createIndex( { "a.loc": 1, "a.qty": 1 }, { unique: true } )
```

제약사항

- hash index 에서는 사용할 수 없다.
- collection 데이터에 uniqueness 를 위반하는 데이터가 있다면 생성할 수 없다.

- replicaset
  - primary 노드에서 수행한다.
- sharded cluster
  - mongos 에서 수생한다.

Partial 인덱스와 Sparse 인덱스

sparse 인덱스

sparse 인덱스는 지정된 필드가 다큐먼트에 존재하는 경우에만 인덱싱을 하는 옵션이다. 예를 들어 필드의 값이 null 인 경우 인덱싱이 되지만 필드 자체가 없는 경우에는 인덱싱이 되지 않는다.

```
db.sparse.createIndex( {birth_date:1}, {sparse:true} )

db.sparse.insert( {"name": "matt"} )
db.sparse.insert( {"name": "lara", "birth_date": null } )
db.sparse.insert( {"name": "cathy", "birth_date": "10-21" } )

// index 를 사용하지 못함
> db.sparse.find( {birth_date: {$exists: false} } )
{ "_id" : ObjectId("6056fec629b907835cae0623"), "name" : "matt" }

// index 를 사용하지만 결과가 완전하지 않음
// sparse index 에 birth_date 가 없는 다큐먼트 정보가 없기 때문에 부정확한 결과가 나온다.
> db.sparse.find( {birth_date: {$exists: false} } ).hint( {birth_date: 1} )
>
```

partial 인덱스

partial 인덱스는 사용자가 조건을 설정하여 다큐먼트를 인덱싱한다. paritialFilterExpression 옵션을 사용하여 사용자가 조건을 설정할 수 있으며 조건에 따라 sparse 인덱스와 동일하게 동작하게 할 수 있다.

- partialFilterExpression
  - equality expressions (i.e. field: value or using the $eq operator)
  - $exists: true expression
  - $gt, $gte, $lt, $lte expressions
  - $type expressions
  - $and operator at the top-level only

> 인덱스 필드가 아닌 필드를 partialFilterExpression 에 사용한 경우 해당필드를 조건에 반드시 사용해야 한다.

TTL 인덱스

특징

- 날짜 타입의 필드를 생상으로 TTL 인덱스를 생성할 수 있다.
- 정해진 expire 시간이 대상 다큐먼트는 삭제 된다.

- 60 간격으로 백그라운드 thread 에 의해 삭제가 시도된다.
- replica set 의 secondary 에서는 primary 에서 삭제되는 시점에 operation 이 적용된다.

제약사항

- 날짜 타입의 필드를 대상으로 index 를 생성할 수 있다.
- compound index 를 생성할 수 없다.
- _id 필드에 생성할 수 없다.
- capped collection 에 생성할 수 없다.
- 기존의 expireAfterSeconds 값을 변경할 수 없다.
- collMod 명령으로 변경할 수 있다.
- 기존의 TTL 인덱스를 제거하고 새로 생성한다.
- 일반 인덱스에 포함된 필드를 대상으로 TTL index 를 생성할 수 없다.

인덱스 콜레이션 (Collation)

mongodb 3.2 까지는 case-sensitive 방식으로 대소문자를 비교하였지만 3.4 부터 collation 이 도입되었다.

- collection 을 만들때 collation 을 지정할 수 있다.
- case insensitive index 를 만들때 collation 을 지정하지 않으면 collection 의 collation 을 상속 받는다.
- collection 에 collation 을 지정하고 index 를 생성한 뒤 쿼리에 collation 이 없을때 collection 의 collation 을 상속 받는다.
- 지정한 collation 은 변경할 수 없다.
- collation 과 다른 collation 으로 검색하는 경우 index 를 사용할 수 없다.

예제

```
db.createCollection("fruit")

db.fruit.createIndex( { type: 1},
                      { collation: { locale: 'en', strength: 2 } } )

db.fruit.insert( [ { type: "apple" },
                   { type: "Apple" },
                   { type: "APPLE" } ] )

// 인덱스를 사용할 수 없다. 결과는 1개가 나온다.
db.fruit.find( { type: "apple" } )

// 인덱스를 사용할 수 있다. 결과는 3개가 나온다.
db.fruit.find( { type: "apple" } ).collation( { locale: 'en', strength: 2 } )

// 인덱스를 사용할 수 없지만 결과는 3개가 나온다.
db.fruit.find( { type: "apple" } ).collation( { locale: 'en', strength: 1 } )


// collection 에 collation 을 지정하는 경우
db.createCollection("names", { collation: { locale: 'en_US', strength: 2 } } )

// index 생성시 collation 을 상속 받는다.
```

```
db.names.createIndex( { first_name: 1 } )

// 데이터 입력
db.names.insert( [ { first_name: "Betsy" },
                   { first_name: "BETSY"},
                   { first_name: "betsy"} ] )

// 쿼리에 collation 을 지정하지 않아도 collection 의 collation 을 따르고
// collation 이 동일한 index 가 있다면 사용할 수 있다.
db.names.find( { first_name: "betsy" } )
```

외래키 (FK)

mongodb 에서는 외래키를 지원하지 않는다.