속도의, 속도에 의한, 속도를 위한 몽고DB

(네이버 컨텐츠검색과 몽고DB)

박근배
지식DB서비스플랫폼
NAVER

CONTENTS

DEVIEW 2019

1. 네이버 검색에서 몽고DB
2. Why MongoDB?
3. MongoDB 속도 올리기 – Index
4. MongoDB 속도 올리기 - ^Index
5. 미운 Index

Introduction

Introduction

박근배

지식DB서비스 플랫폼

- 컨텐츠검색 주제 개발

- 컨텐츠검색 플랫폼 개발

kunbae.park@navercorp.com

1. 네이버 검색에서 몽고DB

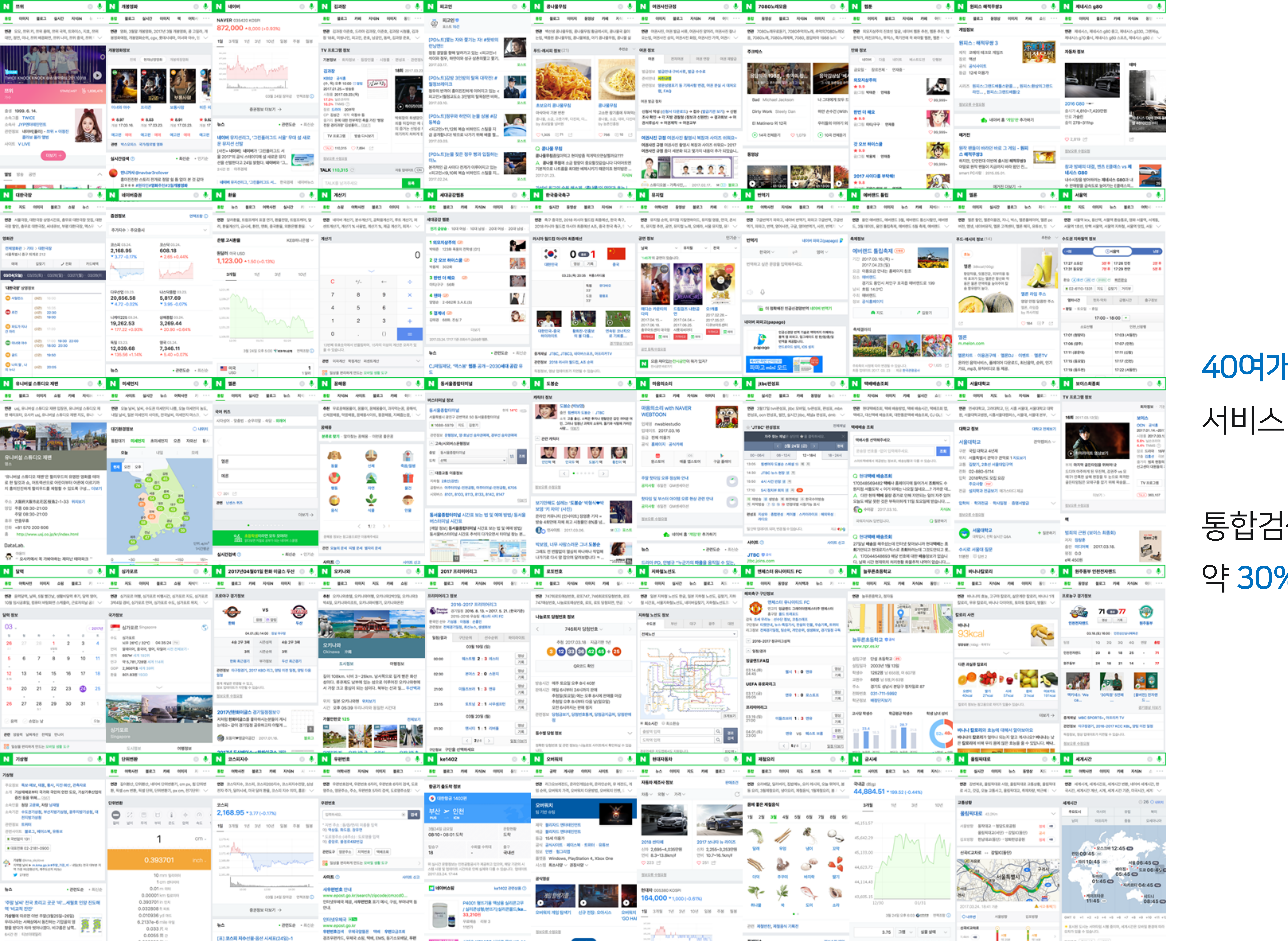
1.1 네이버 컨텐츠검색

인물, 영화, 방송, 날씨, 스포츠 등

다양한 주제의 검색 쿼리 대응.

단순 문서검색이 아닌 고퀄의 검색 결과 제공.

#여러분이 #구글 말고 #네이버검색을 쓰는 이유

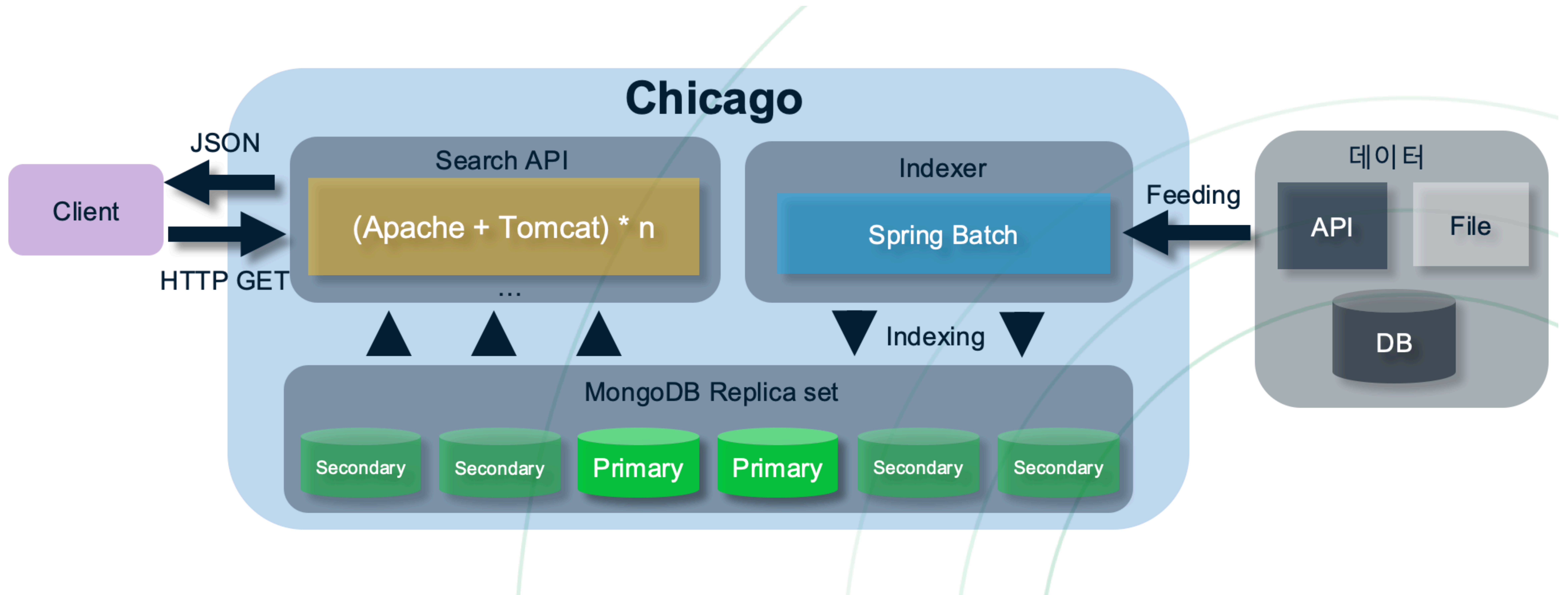


DEVIEW 2019

40여개 주제 250여개 컬렉션으로 서비스 중.

통합검색 전체 질의어 중 약 30% 대응 중.

1.2 네이버 컨텐츠검색과 몽고DB



2. Why MongoDB?

2. Why MongoDB?

FAST + SCALABLE + HIGHLY AVAILABLE = MongoDB

- 네이버 통합검색의 **1초 rule**, 현재 시카고 플랫폼 평균 **응답속도 10ms 미만**.
- 일 **평균 4~5억건 이상** 서버호출, 초당 6천건 수준, 몽고DB는 초당 만건 이상.

3. 몽고DB 속도 올리기 - Index

3-1. MongoDB Index 이해하기

MongoDB는 컬렉션당 **최대 64개**의 인덱스만 생성 가능.

너무 많은 인덱스를 추가하면, 오히려 side effects가 발생한다.

- Frequent Swap.

- Write performance 감소.

3-2. Index Prefix를 사용하자

Index prefixes are **the beginning subsets** of index fields.

```
db.nbaGame.createIndex({teamCode: 1, season: 1, gameDate: 1})
```

Index prefixes of the index

- {teamCode: 1, season: 1}
- {teamCode: 1}

3-2. Index Prefix를 사용하자


```
db.nbaGame.createIndex({teamCode: 1, season: 1, gameDate: 1});
```

```
1. db.nbaGame.find({teamCode: "gsw", season: "2018", gameDate: "20180414"});
2. db.nbaGame.find({teamCode: "gsw", season: "2018"});
3. db.nbaGame.find({teamCode: "gsw"});
```

Covered!

3-2. Index Prefix를 사용하자


```
db.nbaGame.createIndex({teamCode: 1, season: 1});

db.nbaGame.createIndex({teamCode: 1});
```

Not Necessary!

3-2. Index Prefix를 사용하자

DEVIEW 2019

```
db.nbaGame.createIndex({teamCode: 1, season: 1, gameDate: 1});
```

```
db.nbaGame.find({season: “2018”});

db.nbaGame.find({gameDate: “20180414”});

db.nbaGame.find({season: “2018”, gameDate: “20180414”});
```

= Not Covered!

위 모든 쿼리들이 teamCode 필드를 가지고 있지 않음. (not index prefix)

```
db.nbaGame.find({teamCode: “gsw”, gameDate: “20180414”});
```

= Half Covered!

컨디션이 teamCode로 시작하지만 season 필드가 뒤이어 나오지 않음.

3-3. 멀티 소팅

Sort key들은 반드시 인덱스와 **같은 순서로 나열** 되어야만 한다.

```
db.nbaGame.createIndex({gameDate: 1, teamName: 1});
```

```
db.nbaGame.find().sort({gameDate: 1, teamName: 1 });  // covered

db.nbaGame.find().sort({teamName: 1, gameDate: 1 });  // not covered
```

3-3. 멀티 소팅

- single-field 인덱스의 경우, 소팅 **방향을 걱정할 필요 없음**.
- 그러나 compound 인덱스의 경우, 소팅 **방향이 중요함**.

(멀티소팅의 방향은 반드시 index key pattern 또는 index key pattern의 inverse와 일치 해야함)

```
db.nbaGame.createIndex({gameDate: 1, teamName: 1});

db.nbaGame.find().sort({gameDate: 1, teamName: 1 });
db.nbaGame.find().sort({gameDate: -1, teamName: -1 });  // the inverse
db.nbaGame.find().sort({gameDate: 1});   // index prefix
db.nbaGame.find().sort({gameDate: -1});
```

= Covered!

3-3. 멀티 소팅

```
db.nbaGame.createIndex({gameDate: 1, teamName: 1});

db.nbaGame.find().sort({gameDate: -1, teamName: 1 });  // not matched

db.nbaGame.find().sort({gameDate: 1, teamName: -1 });

db.nbaGame.find().sort({teamName : 1});   // not index prefix

db.nbaGame.find().sort({teamName : -1});
```

Not Covered!

3-3. 멀티 소팅


```
db.nbaGame.createIndex({gameDate: 1, teamName: 1});
db.nbaGame.createIndex({gameDate: -1, teamName: 1});
db.nbaGame.createIndex({teamName: 1, gameDate: 1});
db.nbaGame.createIndex({teamName: -1, gameDate: 1});
```

- 위 4개 **인덱스**로 {gameDate, teamName}로 만들 수 있는 아래 12가지 조합들 지원 .
- 아래 12개의 쿼리를 위해 각각 인덱스를 만들 필요가 없다.

```
1. find().sort({gameDate: 1, teamName: 1});     2. find().sort({gameDate: -1, teamName: -1});
3. find().sort({gameDate: 1, teamName: -1});    4. find().sort({gameDate: -1, teamName: 1});
5. find().sort({teamName: 1, gameDate: 1});     6. find().sort({teamName: -1, gameDate: -1});
7. find().sort({teamName: 1, gameDate: -1});    8. find().sort({teamName: -1, gameDate: 1});
9. find().sort({gameDate: 1});                  10. find().sort({gameDate: -1});
11. find().sort({teamName: 1});                 12. find().sort({teamName: -1});
```

4. 몽고DB 속도 올리기 - ^Index

4-1. 하나의 컬렉션을 여러 컬렉션으로 나누자

DEVIEW 2019

mach collection

```
{Job: “nbaGame”, data: … },
{Job: “nbaGame”, data: … },
{job: “nbaTeam”, data: …},
{job: “nbaTeam”, data: …},
{job: “nbaTeam”, data: …},
{job: “mlbGame”, data: …},
{job: “mlbGame”, data: …},

…. 200,000 more
```

```
db.mach.find({
    filter4: “sports”,
    filter5 : “Y”,
    range1: { $gte: 2019}
}).sort({sort1:1}).explain(“executionStats”);
```

```
"executionStats" : {
  "nReturned" : 534,
  "executionTimeMillis" : 412,
  "totalKeysExamined" : 11233,
  "totalDocsExamined" : 11233,
  "executionStages" : { "stage" : IXSCAN
          "nReturned" : 3234,
          ”indexName”: “ix_filter4_sort1_1”
           …
          }
     …
```

4-1. 하나의 컬렉션을 여러 컬렉션으로 나누자

- 하나의 컬렉션이 **너무 많은 문서**를 가질 경우, **인덱스 사이즈가 증가**하고 인덱스 필드의 **cardinality**가 **낮아질 가능성**이 높다.

- 이는 lookup performance에 악영향을 미치고 **slow query**를 유발한다.

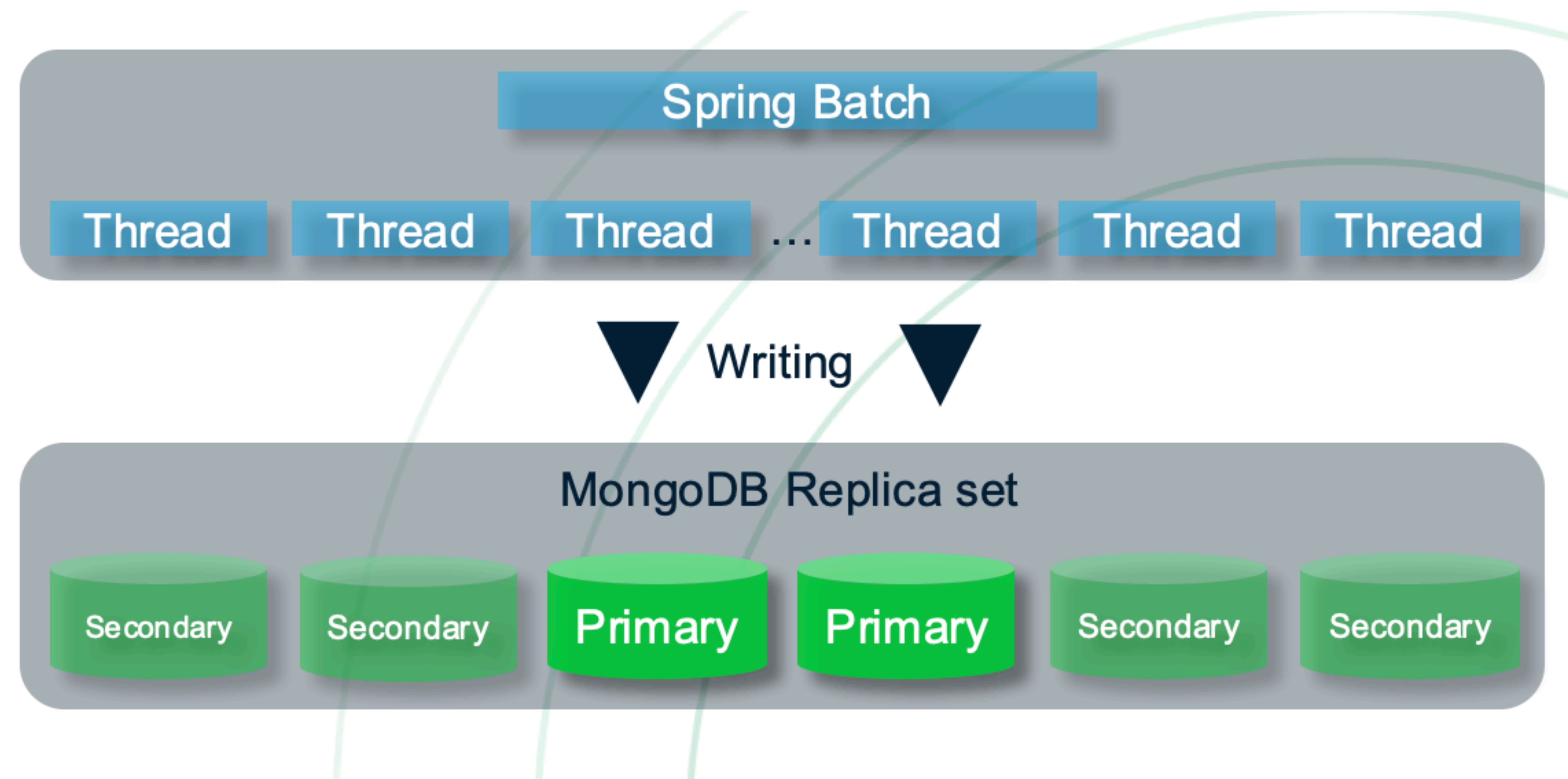
- 한 컬렉션에 너무 많은 문서가 있다면 반드시 **컬렉션을 나눠서** query processor가 **중복되는 인덱스 key**를 look up하는 것을 방지해야 한다.

4-2. 쓰레드를 이용해 대량의 Document를 upsert

- 여러 개의 thread에서 Bulk Operation으로 많은 document를 한번에 write.

- document transaction과 a relation은 지원 안됨.

- Writing time을 획기적으로 개선 가능.



4-3. MongoDB 4.0으로 업그레이드

- 몽고DB 4.0 이전에는 non-blocking secondary read 기능이 없었음.
- Write가 primary에 반영 되고 secondary들에 다 전달 될때까지 secondary는 read를 block해서 데이터가 잘못된 순서로 read되는 것을 방지함.
- 그래서 주기적으로 높은 global lock acquire count가 생기고 read 성능이 저하됨.
- 몽고DB 4.0부터는 data timestamp와 consistent snapshot을 이용해서 이 이슈를 해결함.
- 이게 non-blocking secondary read.

5. 미운 Index

5-1. Slow Queries..

- 컬렉션의 문서수 : 9만건

"오늘 공연"

db.concert.count({serviceCode: { $in: [2, 3] }, startDate: { $lte: 20190722 }, endDate: { $gte: 20190722}});

270ms

"서울 공연"

db.concert. count({ serviceCode: { $in: [2, 3] }, region: "se", isNow: 1});

290ms

5-2. Explain 결과

- 컬렉션의 문서수 : 9만건

"오늘 공연"

```
db.concert.count({serviceCode: { $in: [ 2, 3 ] }, startDate: { $lte: 20190722 }, endDate: { $gte: 20190722}});
```

=> {serviceCode:1, weight:1} 인덱스를 이용 (key examined : 88226)

"서울 공연"

```
db.concert. count({ serviceCode: { $in: [ 2, 3 ] }, region: "se", isNow: 1});
```

=> {region: 1, weight: 1} 인덱스를 이용 (key examined : 25322)

5-3. 이게 최선일까?

"오늘 공연"

```
db.concert.count({serviceCode: { $in: [ 2, 3 ] }, startDate: { $lte: 20190722 }, endDate: { $gte: 20190722}});
```

=> {serviceCode:1, weight:1} 인덱스를 이용 (key examined : 88226)

결과값은 123.

123개의 문서를 찾기 위해 **8만개의 문서를 탐색** 한다???

serviceCode가 2,3 인 8만개의 인덱스 키를 훑어보면서,

메모리에서 startDate, endDate의 범위 검색을 실행.

사실 startDate, endDate부터 실행하고 serviceCode를 뒤지면 123개만 탐색하면 가능함.. 근데 왜..?

5-3. 이게 최선일까?

"서울 공연"

db.concert. count({ serviceCode: { $in: [2, 3] }, region: "se", isNow: 1});

=> {region: 1, weight: 1} 인덱스를 이용 (key examined : 25322)

결과값은 75.

75개 문서를 찾기 위해 2만 5천개를 탐색 한다???

region이 "se"인 2만 5천개의 인덱스를 훑어 보면서,

메모리에서 isNow이 1이고 serviceCode가 2, 3인 문서를 검색.

마찬가지로 isNow 부터 검색하고 region, serviceCode를 검색하면 200여개만 탐색하면 됨.. 근데 왜..?

5-4. 인공지능 보단 인간지능

"오늘 공연"

```
db.concert.count({serviceCode: { $in: [ 2, 3 ] }, startDate: { $lte: 20190722 }, endDate: { $gte: 20190722});
```

~~=> {serviceCode:1, weight:1} 인덱스를 이용 (key examined : 88226)~~

{serviceCode: 1, startDate:1, endDate: 1} 인덱스를 만듬.

예상 시나리오:

새로 만든 Index를 타게 되면 serviceCode: 2,3이면서 startDate, endDate 범위검색을 Index에서 끝냄.

5-4. 인공지능 보단 인간지능

결과는?

5-4. 인공지능 보단 인간지능

응. 안타.

5-4. 인공지능 보단 인간지능

"서울 공연"

db.concert. count({ serviceCode: { $in: [2, 3] }, region: "se", isNow: 1});

~~=> {region: 1, weight: 1} 인덱스를 이용 (key examined : 25322)~~

{serviceCode: 1, region:1, isNow: 1} 인덱스를 만듬.

예상 시나리오:

새로 만든 Index를 타게 되면 serviceCode, region, isNow 조건 검색을 Index에서 끝냄.

5-4. 인공지능 보단 인간지능

결과는?

5-4. 인공지능 보단 인간지능

응. 너도 안타.

5-4. 인공지능 보단 인간지능

hint를 이용해 강제로 인덱스를 태워봤더니….

```
db.concert.count({
    serviceCode: { $in: [ 2, 3 ]},  startDate: { $lte: 20190722 }, endDate: { $gte: 20190722},
    {hint: ”serviceCode_1_startDate_1_endDate_e_1”}
);
```

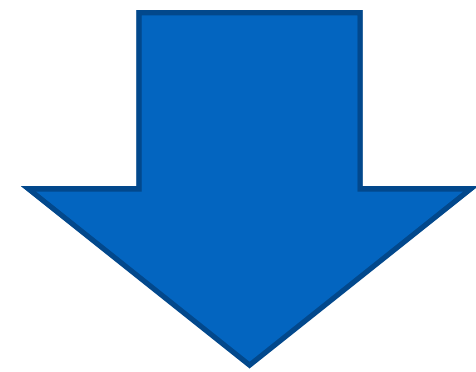
270ms -> 3ms

```
db.concert. count({
    serviceCode: { $in: [ 2, 3 ] }, region : "se”, isNow : 1},
    {hint: “region_1_isNow_1_serviceCode_1”}
);
```

290ms -> 1ms

5-5. Index가 미운짓을 하는 이유

왜 자꾸 엉뚱한 인덱스를 타는걸까? 에서 의문시작.



몽고 DB는 쿼리가 들어왔을때 **어떻게 최선의 쿼리플랜을 찾을까?**

여러개의 쿼리 플랜들을 모두 실행해 볼 수도 없고…

5-5. Index가 미운짓을 하는 이유

DEVIEW 2019

답은 Query Planner 에서 찾을 수 있었음.

이전에 실행한 쿼리 플랜을 캐싱 해놓음.

캐싱된 쿼리 플랜이 없다면

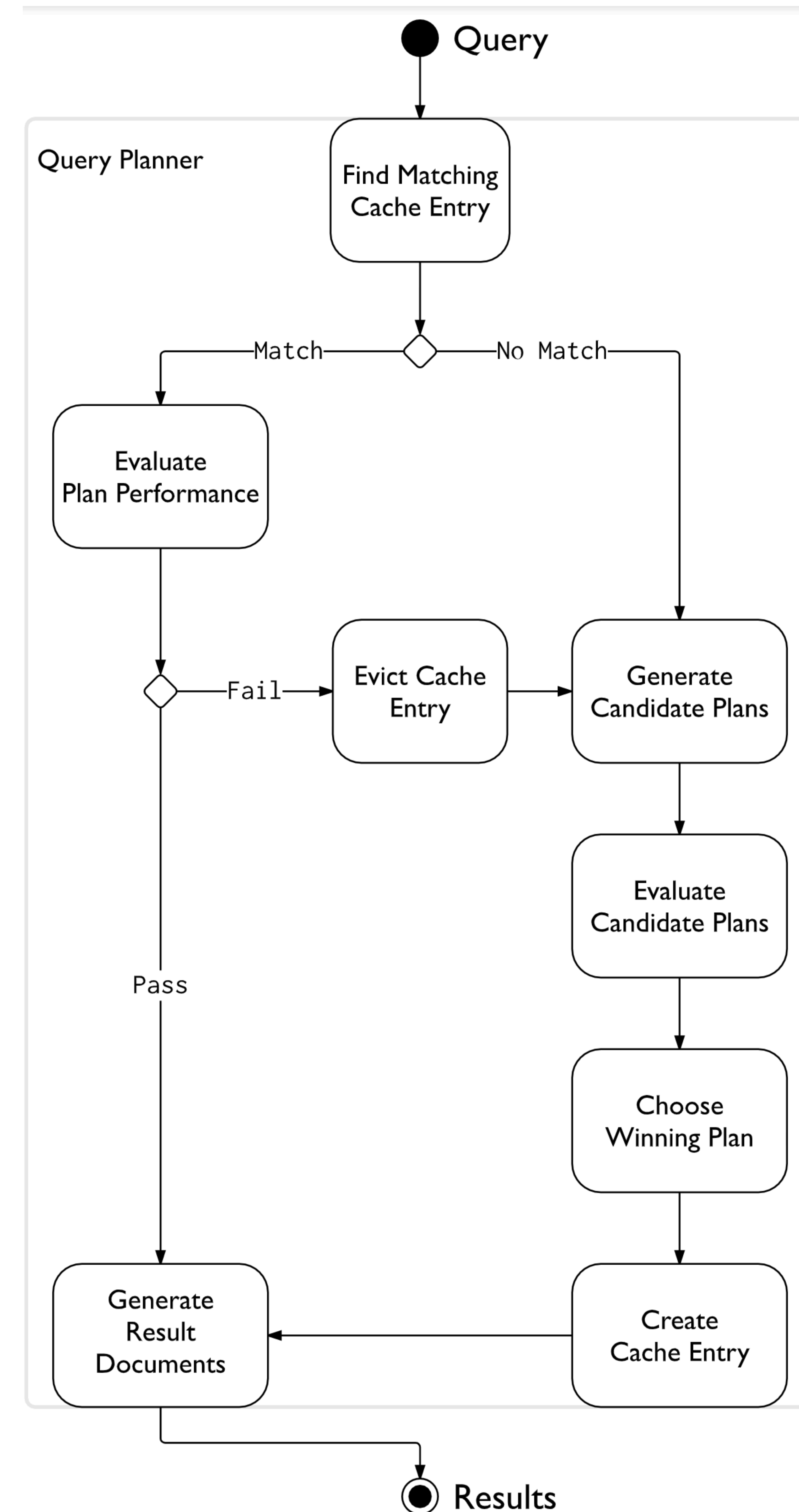
가능한 모든 쿼리 플랜들을 조회해서

첫 batch(101개)를 가장 좋은 성능으로

가져오는 플랜을 다시 캐싱함.

성능이 너무 안좋아지면 위 작업을 반복.

* 캐싱의 사용 조건은 같은 Query Shape 일 경우



5-5. Index가 미운짓을 하는 이유

DEVIEW 2019

결국 { serviceCode: 1, weight:1 } 인덱스를 사용하더라도

```
db.concert.count({
    serviceCode: { $in: [ 2, 3 ]}, startDate: { $lte: 20190722 }, endDate: { $gte: 20190722}
});
```

이 쿼리는 **전체 조회가 필요해** 느릴 수 있지만 (key examined : 88226)

```
db.concert.find({
    serviceCode: { $in: [ 2, 3 ]}, startDate: { $lte: 20190722 }, endDate: { $gte: 20190722}
}).limit(101);
```

101개의 오브젝트만 가져온다면 순식간에 가져옴.

(101개를 찾자 마자 return 하므로 key examined : 309)

5-5. Index가 미운짓을 하는 이유

그래서 Query Planner가 count() 쿼리의 Query Shape로

첫번째 batch를 가져오는 성능을 테스트 했을때

{serviceCode: 1, weight:1} 같은 엄한 인덱스가 제일 좋은 성능을 보일 수 있는거임.

* 만약에 동점이 나올 경우, in-memory sort를 하지 않아도 되는 쿼리 플랜을 선택함.
(몽고DB는 32MB가 넘는 결과값들에 대해선 in-memory sort가 불가능 하므로)

5-6. 솔로몬의 선택

Hint 이용	VS	엄한 인덱스를 지우기
- 확실한 쿼리플랜 보장 - 더 효율적인 인덱스가 생겨도 강제 고정 - 데이터 분포에 대한 정보를 계속 follow 해야함 - 32MB넘는 응답결과를 sort해야 할 경우 에러		- 데이터에 따라 더 효율적인 인덱스가 생기면 자동 대응 - 또다른 엄한 케이스가 생길 수 있음 - 삭제로 인해 영향 받는 다른 쿼리가 생길 수 있음

5-6. 솔로몬의 선택

Hint 이용

- 확실한 쿼리플랜 보장
- 더 효율적인 인덱스가 생겨도 강제 고정
- 데이터 분포에 대한 정보를 계속 follow 해야함
- 32MB넘는 응답결과를 sort해야 할 경우 에러

VS

PICKED!

엄한 인덱스를 지우기

- 데이터에 따라 더 효율적인 인덱스가 생기면 자동 대응
- 또다른 엄한 케이스가 생길 수 있음
- 삭제로 인해 영향 받는 다른 쿼리가 생길 수 있음

Q & A

Thank You