CHAPTER 1

About

...
2.1 Cascading Multi-Clause Queries

Author  Shane R. Spencer <shane@bogomip.com>
Date  Sun Oct 20 01:14:35 UTC 2013

Contents

- Cascading Multi-Clause Queries
  - Preface
  - Summary
  - Data
    * Sample Twitter Post
    * Indexes
  - The Problem
  - The Solution
    * Multi-Index Support
    * Sorting Clause Results
  - Conclusion

2.1.1 Preface

In MongoDB multi-clause queries are done by way of the $or logical query operator.

Cascading refers to the application level preference of a multi-clause query so that certain index regions and table scans are processed in a specific order. Furthermore, the application can choose to use the cursor.limit() method that allows multi-clause queries to exit early without processing all the clauses if the limit is reached. What a wonderful optimization for large data sets.

Here is a 2 clause query from the official MongoDB documentation where price is part of the first query clause and sale is part of the second while qty further filters the results of each:

```javascript
db.inventory.find({
  "$or": [{
    price: 1.99
  }, {
    sale: true
  }],
  qty: {}
})
```
Simply put... if there are 100 inventory items with a price of 1.99 that match the qty filter and we limit the overall query to 100 documents then the price clause will completely satisfy the query and any further query processing will be ignored. This is a good example of how cascading is being put to use by returning documents in clause processed order.

### 2.1.2 Summary

I inadvertently found out about multi-clause queries while examining a well crafted query that I was feeling incredibly hopeful about. For this query I was looking to create a single stream of documents by querying different values of a location field and I ended up using $or to see how it would react. The goal for me was to first query documents with a specific location and then all the surrounding cities in an order I had determined before making the query.

To take advantage of this I knew I would need to feed the query an ordered set of locations which I would pregenerate based on my own algorithms based on the application users preferences.

Logically, $or performed multiple queries in order where I was wrongly thinking of it as a post-filter for a single query. Amazed, it finally clicked that there was an opportunity be able extend a single query into a concatenation of multiple queries with a single execution of db.collection.find().

### 2.1.3 Data

This article focuses on using a sample stream of geographically referenced Twitter posts using the Twitter Streaming API which is a streaming JSON feed that easily imports into MongoDB.

#### Sample Twitter Post

Geographically referenced Twitter posts contain location information through the place field which focuses on the nearest city and state information for the coordinates field that defines where on earth the post was approximately made from.

```javascript
db.tweets.findOne({
    "place.full_name": "Los Angeles, CA"
}, {
    "text": true,
    "user.screen_name": true,
    "coordinates": true,
    "place.full_name": true,
    "place.country": true
});
```

```json
{
    "_id": ObjectId("52647c32b7c03befed384f00"),
    "text": "Time is going by so fast.",
    "user": {
        "screen_name": "DoctorWhomz"
    },
    "coordinates": {
        "type": "Point",
```
Indexes

The following compound index is in place for test queries that will be looking at the geographic information within each post.

```javascript
// place.country_1_place.full_name_1
db.tweets.ensureIndex({
    "place.country": 1,
    "place.full_name": 1
});
```

### 2.1.4 The Problem

Based on the applications users preference we want to query all twitter users that have more than 500 followers and have made a post recently from one major city to the next and then eventually the entire country.

The user has the following preference:

- Los Angeles, CA
- Manhattan, NY
- Philadelphia, PA
- Chicago, IL
- Houston, TX
- and finally simply United States

### 2.1.5 The Solution

Building a query to deal with explicitly defined ordering using $or is relatively easy since we know exactly what we want to search for. From the API standpoint the language needs to append dictionary or SON objects to the $or field in order. For the following example query we will turn on cursor.explain with `verbose` toggled on.

Since we used $or we will have a clauses array that specifies the clauses and the query plans being used.

```javascript
db.tweets.find({
    "$or": [{
        "place.country": "United States",
        "place.full_name": "Los Angeles, CA",
    }, {
        "place.country": "United States",
        "place.full_name": "Manhattan, NY",
    }],
});
```
}),

```json
{
  "clauses": [{
    "allPlans": [{
      "cursor": "BtreeCursor place.country_1_place.full_name_1",
      "n": 38,
      "nscannedObjects": 38,
      "nscanned": 38,
      "indexBounds": {
        "place.country": [
          "United States",
          "United States"
        ],
        "place.full_name": [
          "Los Angeles, CA",
          "Los Angeles, CA"
        ]
      }
    }]
  }, {  
    "allPlans": [{
      "cursor": "BtreeCursor place.country_1_place.full_name_1",
      "n": 25,
      "nscannedObjects": 25,
      "nscanned": 25,
      "indexBounds": {
        "place.country": [
          "United States",
          "United States"
        ],
        "place.full_name": [
          "Manhattan, NY",
          "Manhattan, NY"
        ]
      }
    }]
  }]
})
```

// Shortened and Simplified
{
  "clauses": [{
    "allPlans": [{
      "cursor": "BtreeCursor place.country_1_place.full_name_1",
      "n": 38,
      "nscannedObjects": 38,
      "nscanned": 38,
      "indexBounds": {
        "place.country": [
          "United States",
          "United States"
        ],
        "place.full_name": [
          "Los Angeles, CA",
          "Los Angeles, CA"
        ]
      }
    }]
  }, {  
    "allPlans": [{
      "cursor": "BtreeCursor place.country_1_place.full_name_1",
      "n": 25,
      "nscannedObjects": 25,
      "nscanned": 25,
      "indexBounds": {
        "place.country": [
          "United States",
          "United States"
        ],
        "place.full_name": [
          "Manhattan, NY",
          "Manhattan, NY"
        ]
      }
    }]
  }]
}
That’s a lot of documents and since we are working with potentially live Twitter data we know it’s going to grow like crazy. Thankfully we can request that the user do some pagination if they want to see all the documents. The above information shows that Los Angeles, CA has 38 tweet documents associated with it and Manhattan, NY has 25. If the application limits each page to 50 documents per page the cursor would only fetch documents from the first two clauses for the first page.

```javascript
db.tweets.find({
  "$or": [
    {
      "place.country": "United States",
      "place.full_name": "Los Angeles, CA",
    },
    {
      "place.country": "United States",
      "place.full_name": "Manhattan, NY",
    },
    {
      "place.country": "United States",
      "place.full_name": "Philadelphia, PA",
    },
    {
      "place.country": "United States",
      "place.full_name": "New York, NY",
    }
  ]
})
```


```javascript
// Shortened and Simplified
{
    "clauses" : [
        { "allPlans" : [
            {
                "cursor" : "BtreeCursor place.country_1_place.full_name_1",
                "n" : 38,
                "nscannedObjects" : 38,
                "nscanned" : 38,
                "indexBounds" : {
                    "place.country" : [
                        "United States",
                        "United States"
                    ],
                    "place.full_name" : [
                        "Los Angeles, CA",
                        "Los Angeles, CA"
                    ]
                }
            }
        },
        { "allPlans" : [
            {
                "cursor" : "BtreeCursor place.country_1_place.full_name_1",
                "n" : 12,
                "nscannedObjects" : 12,
                "nscanned" : 12,
                "indexBounds" : {
                    "place.country" : [
                        "United States",
                        "United States"
                    ],
                    "place.full_name" : [
                        "Manhattan, NY",
                        "Manhattan, NY"
                    ]
                }
            }
        }
    ]
}
```
I have a lot of appreciation for milliseconds: 0.

As previously stated, the user wants to include only documents posted by individuals that have more than 500 followers. We can do this one of two ways depending on how flexible we want this query.

```javascript
db.tweets.find({
    "$or": [{
        "place.country": "United States",
        "place.full_name": "Los Angeles, CA",
    }, {
        "place.country": "United States",
        "place.full_name": "Manhattan, NY",
    }, {
        "place.country": "United States",
        "place.full_name": "Philadelphia, PA",
    }, {
        "place.country": "United States",
        "place.full_name": "Chicago, IL",
    }, {
        "place.country": "United States",
        "place.full_name": "Houston, TX",
    }],
    "user.followers_count": { "$gte": 500 },
}).limit(50).explain( verbose = true )
```

```javascript
db.tweets.find({
    "$or": [{
        "place.country": "United States",
        "place.full_name": "Los Angeles, CA",
        "user.followers_count": { "$gte": 500 },
    }, {
        "place.country": "United States",
        "place.full_name": "Manhattan, NY",
        "user.followers_count": { "$gte": 500 },
    }, {
        "place.country": "United States",
        "place.full_name": "Philadelphia, PA",
        "user.followers_count": { "$gte": 500 },
    }, {
        "place.country": "United States",
        "place.full_name": "Chicago, IL",
        "user.followers_count": { "$gte": 500 },
    }, {
        "place.country": "United States",
        "place.full_name": "Houston, TX",
        "user.followers_count": { "$gte": 500 },
    }],
}).limit(50).explain( verbose = true )
```
The latter query allows us to change `user.followers_count` to match any limit the user requests for each region. Perhaps they want to scan the country for any individuals with over 10000 followers.

**Multi-Index Support**

Each clause can rely on a different indexes. However, there’s no method of applying a `cursor.hint()` to individual clauses.

For instance if you wanted to use a sparse index in the first clause but wanted to use a compound index for the rest of them then you would want to specifically query around whatever fields are involved with the index you want to use.

```javascript
// user.screen_name_1
db.tweets.ensureIndex({
  "user.screen_name": 1,
}, {
  "sparse": true
});

db.tweets.find({
  "$or": [{
    "user.screen_name": "DoctorWhomz",
  }, {
    "place.country": "United States",
    "place.full_name": "Houston, TX",
  }, {
    "place.country": "United States"
  }]
}).explain(verbose = true);
```

In the example above the following indexes will be used in order:

- `user.screen_name_1` (sparse)
- `place.country_1.place.full_name_1`
- `place.country_1.place.full_name_1`

**Sorting Clause Results**

You shouldn’t use `cursor.sort()` with this technique since it would defeat the tiering of data each clause helps establish and you can easily run into a situation where you are sorting result sets larger than the `_MongoDB sort buffer`. Instead consider the following: retrieve all of the results and sort them in your application code by whatever means necessary or plan on using compound indexes and encourage MongoDB to use the indexes ordering as the sorting method for each clauses result set.

I personally prefer the latter option since it forces me to test the indexes and make sure they are optimal in the process. Here’s an example where I am creating another index that will ultimately have `user.screen_name` in sorted order after `place.full_name`.

```
```
In order to make sure a clause uses this index since it shares its first two keys with another index the query conditions simply need to require that `user.screen_name` be $gte the lowest possible string value. This is a sneaky way of including the `user.screen_name` into the query planners decision making.

Now there is a higher if not totally positive chance that _MongoDB will choose the index `place.country` place.full_name place.user.screen_name` rather than simply `place.country` place.full_name` and the results would be returned in the expected order.

### 2.1.6 Conclusion

_MongoDB definitely encourages developers to think outside of the relational database box and create some clever query optimizations that allow MongoDB to operate at its peak performance. This includes finding ways to reduce table scans, discovering the right index for the job, and using some lesser known optimizations like the $or logical query operator to do what I have been calling Cascading Multi-Clause Queries._