

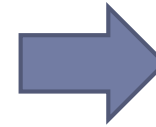
MongoDB – Aggregation & Performance

Aggregation

- ▶ Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.
- ▶ MongoDB provides three ways to perform aggregation:
 - ▶ The aggregation pipeline
 - ▶ The map-reduce function
 - ▶ Single purpose aggregation methods
`(db.collection.count(), db.collection.group(), db.collection.distinct())`

SQL vs Aggregate Example

id	name	category	manufacturer	price
1	iPad	Tablet	Apple	800
2	Nexus	Phone	Google	500
3	iPhone	Phone	Apple	600
4	iPadPro	Tablet	Apple	900

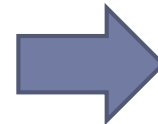


manufacturer	count
Apple	3
Google	1

```
select manufacturer, count(*) from products group by manufacturer
```

```
{ _id: 1, name: 'iPad', category: 'Tablet', manufacturer: 'Apple', price: 800 }  
{ _id: 2, name: 'Nexus', category: 'Phone', manufacturer: 'Google', price: 500 }  
{ _id: 3, name: 'iPhone', category: 'Phone', manufacturer: 'Apple', price: 600 }  
{ _id: 4, name: 'iPadPro', category: 'Tablet', manufacturer: 'Apple', price: 900 }
```

```
db.products.aggregate([  
  {$group: {  
    _id:"$manufacturer",  
    num_products:{$sum:1} }  
  ]})
```



```
{ _id: 'Apple', num_products: 3 }  
{ _id: 'Google', num_products: 1 }
```

```
]
```

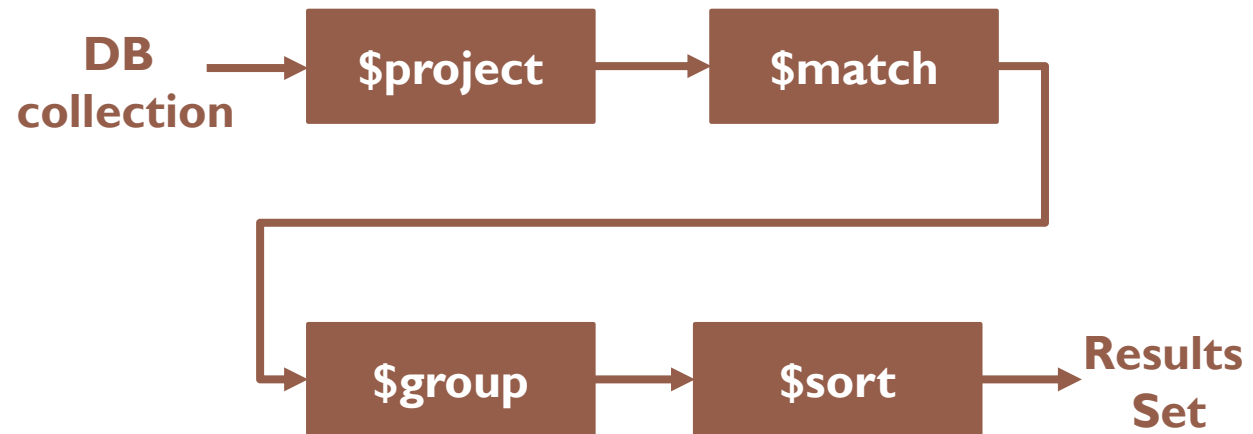
Aggregation Pipeline

- ▶ The aggregation framework is built on the concept of **data processing pipelines**. Documents enter a multi-stage pipeline that transforms the documents into an **aggregated result**.
- ▶ The pipeline provides efficient data aggregation using native operations within MongoDB.
- ▶ The aggregation pipeline can operate on a sharded collection.
- ▶ The aggregation pipeline can use indexes to improve its performance during some of its stages (only if done at the beginning of the aggregation pipeline).
- ▶ Pipeline stages can appear multiple times in the pipeline.

Aggregation Pipeline Stages

- ▶ \$group
- ▶ \$project
- ▶ \$match
- ▶ \$sort
- ▶ \$limit
- ▶ \$skip
- ▶ \$unwind
- ▶ \$out
- ▶ \$lookup
- ▶ \$redact
- ▶ \$geoNear

There is a 100 MB limit for any pipeline stage.



Can be a cursor or saved in a collection

More Stage operators

\$group

- ▶ Think of **\$group** as an **UPSERT** stage, where it'll insert if **_id** is not found, or update when available.

```
db.products.aggregate([
  {$group: { _id:"$manufacturer",
             num_products:{$sum:1} } }
])
```



```
{ _id: 'Apple',  num_products: 3 }
{ _id: 'Google', num_products: 1 }
```

```
db.products.aggregate([
  {$group: { _id: { 'manufacturer':"$manufacturer"},
             num_products:{$sum:1} } }
])
```

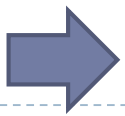


```
{ "_id" : { "manufacturer" : "Google" },
  "num_products" : 1 }
{ "_id" : { "manufacturer" : "Apple" },
  "num_products" : 3 }
```

When we want to use a key as **DATA** in the right side to read its value we must use **\$sign** with the name. We don't need to do that when using it at the left side as it is just a label.

Compound Grouping

```
db.products.aggregate([
  {$group: { _id: { "manufacturer":"$manufacturer", "category" : "$category"},
             num_products:{$sum:1} } }
])
```



```
{ "_id" : { "manufacturer" : "Apple", "category" : "Phone" }, "num_products" : 1 }
{ "_id" : { "manufacturer" : "Google", "category" : "Phone" }, "num_products" : 1 }
{ "_id" : { "manufacturer" : "Apple", "category" : "Tablet" }, "num_products" : 2 }
```

Aggregation Expressions with \$group

`$sum`, `$avg`, `$min`, `$max`, `$push`, `$addToSet`, `$first`, `$last`

```
{ _id: 1, name: 'iPad', category: 'Tablet', manufacturer: 'Apple', price: 800 }
{ _id: 2, name: 'Nexus', category: 'Phone', manufacturer: 'Google', price: 500 }
{ _id: 3, name: 'iPhone', category: 'Phone', manufacturer: 'Apple', price: 600 }
{ _id: 4, name: 'iPadPro', category: 'Tablet', manufacturer: 'Apple', price: 900 }
```

```
db.products.aggregate([
  {$group: { _id: "$manufacturer",
             sum_prices:{$sum:"$price"} } }
])
```

```
{ _id: 'Apple', sum_prices: 2300 }
{ _id: 'Google', sum_prices: 500 }
```

```
db.products.aggregate([
  {$group: { _id: "$category",
             avg_price:{$avg:"$price"} } }
])
```

```
{ "_id" : "Phone", "avg_price" : 550 }
{ "_id" : "Tablet", "avg_price" : 850 }
```

```
db.products.aggregate([
  {$group: { _id: "$manufacturer",
             maxprice:{$max:"$price"} } }
])
```

```
{ _id: 'Apple', maxprice: 900 }
{ _id: 'Google', maxprice: 500 }
```

Notice that `$max` and `$min` do not give us much information about the document. To solve this, we can sort the results on one stage and then retrieve `$first` or `$last` in second stage. (refer to the example with `$sort`)

Aggregation Expressions with \$group

```
{ _id: 1, name: 'iPad', category: 'Tablet', manufacturer: 'Apple', price: 800 }  
{ _id: 2, name: 'Nexus', category: 'Phone', manufacturer: 'Google', price: 500 }  
{ _id: 3, name: 'iPhone', category: 'Phone', manufacturer: 'Apple', price: 600 }  
{ _id: 4, name: 'iPadPro', category: 'Tablet', manufacturer: 'Apple', price: 900 }
```

```
db.products.aggregate([  
  {$group: { _id: "$manufacturer",  
             categories:{$addToSet:"$category"} } }  
])
```

```
{ _id: 'Apple', categories: ['Tablet', 'Phone'] }  
{ _id: 'Google', categories: ['Phone'] }
```

```
db.products.aggregate([  
  {$group: { _id: { "maker":"$manufacturer" },  
             categories:{$push:"$category"} } }  
])
```

```
{ _id: { "maker":"Apple" }, categories: ['Tablet', 'Phone', 'Tablet'] }  
{ _id: { "maker":'Google' }, categories: ['Phone'] }
```


\$project

- Use it to remove a key, add new key, rephrase a key or with some simple functions: **\$toUpper** and **\$toLower** for strings, **\$add** and **\$multiply** for numbers.

```
{ _id: 1, name: 'iPad', category: 'Tablet', manufacturer: 'Apple', price: 800 }  
{ _id: 2, name: 'Nexus', category: 'Phone', manufacturer: 'Google', price: 500 }  
{ _id: 3, name: 'iPhone', category: 'Phone', manufacturer: 'Apple', price: 600 }  
{ _id: 4, name: 'iPadPro', category: 'Tablet', manufacturer: 'Apple', price: 900 }
```

```
db.products.aggregate([  
  {$project: { _id:0,  
               'maker': {$toLower:'$manufacturer'},  
               'details': {'category':'$category',  
                           'price':{$multiply:['$price',10]} },  
               'item':'$name' } } ])
```

```
{ maker: 'apple', details: { category: 'Tablet', price: 8000 }, item: 'iPad'}  
{ maker: 'google', details: { category: 'Phone', price: 5000 }, item: 'Nexus'}  
{ maker: 'apple', details: { category: 'Phone', price: 6000 }, item: 'iPhone'}  
{ maker: 'apple', details: { category: 'Tablet', price: 9000 }, item: 'iPadPro'}
```

\$match: Use it to filter the collection

```
{ "_id" : "52556", "city" : "FAIRFIELD", "loc" : [ -91.957611, 41.003943 ], "pop" : 12147, "state" : "IA" }
{ "_id" : "52601", "city" : "BURLINGTON", "loc" : [ -91.116972, 40.808665 ], "pop" : 30564, "state" : "IA" }
{ "_id" : "52641", "city" : "MOUNT PLEASANT", "loc" : [ -91.56142699999999, 40.964573 ], "pop" : 11113, "state" : "IA" }
{ "_id" : "52241", "city" : "CORALVILLE", "loc" : [ -91.590608, 41.693666 ], "pop" : 12646, "state" : "IA" }
{ "_id" : "52240", "city" : "IOWA CITY", "loc" : [ -91.51119199999999, 41.654899 ], "pop" : 25049, "state" : "IA" }
{ "_id" : "52245", "city" : "IOWA CITY", "loc" : [ -91.51506999999999, 41.664916 ], "pop" : 21140, "state" : "IA" }
{ "_id" : "52246", "city" : "IOWA CITY", "loc" : [ -91.56688200000001, 41.643813 ], "pop" : 22869, "state" : "IA" }
```

```
db.zipcodes.aggregate([
  { $match: { state: "IA" } },
  { $group: { _id: "$city",
              population: { $sum: "$pop" },
              zip_codes: { $addToSet: "$_id" } } },
  { $project: { _id: 0,
                city: "$_id",
                population: 1,
                zip_codes: 1 } }
])
```

1. Filter the zipcode collection and leave Iowa state entries
2. Group results by city, calculate the population, and add new field contains zipcode array for each city
3. Remove `_id`, project only city name, population and zipcodes.

```
{ "city" : "FAIRFIELD", "population" : 12147, "zip_codes" : ["52556"] }
{ "city" : "BURLINGTON", "population" : 30564, "zip_codes" : ["52601"] }
{ "city" : "MOUNT PLEASANT", "population" : 11113, "zip_codes" : ["52641"] }
{ "city" : "CORALVILLE", "population" : 12646, "zip_codes" : ["52241"] }
{ "city" : "IOWA CITY", "population" : 69058, "zip_codes" : ["52240", "52245", "52246"] }
```

\$sort

- ▶ Sorting can be done on Disk, or in Memory (default).
- ▶ When sort is an early stage it may use indexes.
- ▶ It can be used before/after grouping.

```
db.zips.aggregate([
    {$match: { state:"IA" } },
    {$group: { _id: "$city", population: {$sum:"$pop"} } },
    {$project: { _id: 0, city: "$_id", population: 1, } },
    {$sort: { population:-1 } }
])
```

Using \$first and \$last with \$sort

- It's useful to use `$first` and `$last` after `$sort`, otherwise their result will be arbitrary. *(only available in the \$group stage)*

```
{ "_id" : "52556", "city" : "FAIRFIELD", "loc" : [ -91.957611, 41.003943 ], "pop" : 12147, "state" : "IA" }
{ "_id" : "52601", "city" : "BURLINGTON", "loc" : [ -91.116972, 40.808665 ], "pop" : 30564, "state" : "IA" }
{ "_id" : "52641", "city" : "MOUNT PLEASANT", "loc" : [ -91.56142699999999, 40.964573 ], "pop" : 11113, "state" : "IA" }
{ "_id" : "52241", "city" : "CORALVILLE", "loc" : [ -91.590608, 41.693666 ], "pop" : 12646, "state" : "IA" }
{ "_id" : "52240", "city" : "IOWA CITY", "loc" : [ -91.51119199999999, 41.654899 ], "pop" : 25049, "state" : "IA" }
{ "_id" : "52245", "city" : "IOWA CITY", "loc" : [ -91.51506999999999, 41.664916 ], "pop" : 21140, "state" : "IA" }
{ "_id" : "52246", "city" : "IOWA CITY", "loc" : [ -91.56688200000001, 41.643813 ], "pop" : 22869, "state" : "IA" }
```

```
db.zips.aggregate([
  {$group: { _id: {state:"$state", city:"$city"},
            population: {$sum:"$pop"}, } },
  {$sort: {"_id.state":1, "population":-1} },
  {$group: { _id:"$_id.state",
            city: {$first: "$_id.city"},
            population: {$first:"$population"} } },
  {$sort: {"_id":1} }
])
```

Get the largest city in every state:

1. Get the population of every city in every state
2. Sort by state, population
3. Group by state, get the first item in each group
4. Now sort by state again (Group might change the order)

\$skip and \$limit

- ▶ It's useful to use **\$skip** and **\$limit** after **\$sort**, otherwise the result will be arbitrary.

```
db.zips.aggregate([
  {$match: { state:"IA" } },
  {$group: { _id: "$city",
             population: {$sum:"$pop"} } },
  {$project: { _id: 0,
               city: "$_id",
               population: 1, } },
  {$sort: { population:-1 } },
  {$skip: 10},
  {$limit: 5}
])
```

- ▶ **Note:** The order of `.sort()`, `.skip()` and `.limit()` methods when applied to a collection cursor does not matter. While the order of `$sort`, `$skip` and `$limit` aggregation stages matters.

Aggregation Framework Limitations

- ▶ There is 100 MB RAM limit for every stage in the pipeline, If you want to work with bigger data use Disk instead(`allowDiskUse: true`)
- ▶ If you want to save the results to a collection using `$out`, remember that there is 16 MB limit size for every document (it's not recommended, better to return a cursor and work with it)
- ▶ Aggregation will work fine in sharded environments but the results from `$group` and `$sort` will be sent back to the first sharded DB.
- ▶ It's not recommended to use Map/Reduce functionalities in MongoDB, use Aggregation Framework. Hadoop is better at handling Map/Reduce tasks, you may want to consider using [Hadoop Connector](#)

Performance

- ▶ When we talk about performance we think adding memory, replacing the hard drive with SSD or using more powerful CPU (vertical scaling).
- ▶ But what really affects performance is our algorithm and the logic we use in structuring our DB!
- ▶ Understanding indexes leads to a better code and performance.

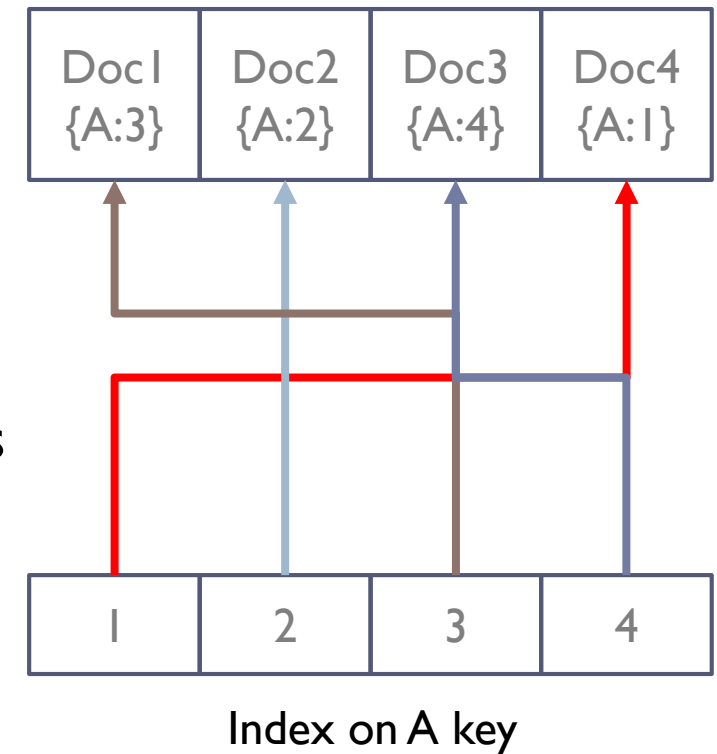


Single Field Indexes

- ▶ When we call `find()`, a **full table/collection scan** will happen because our collection is not sorted which leads to slow performance. That's why we create an Index to boost the performance of our operations (*find, update, sort.. etc*)
- ▶ MongoDB provides complete support for indexes on any field in a collection of documents. By default, all collections have an index on the `_id` field.

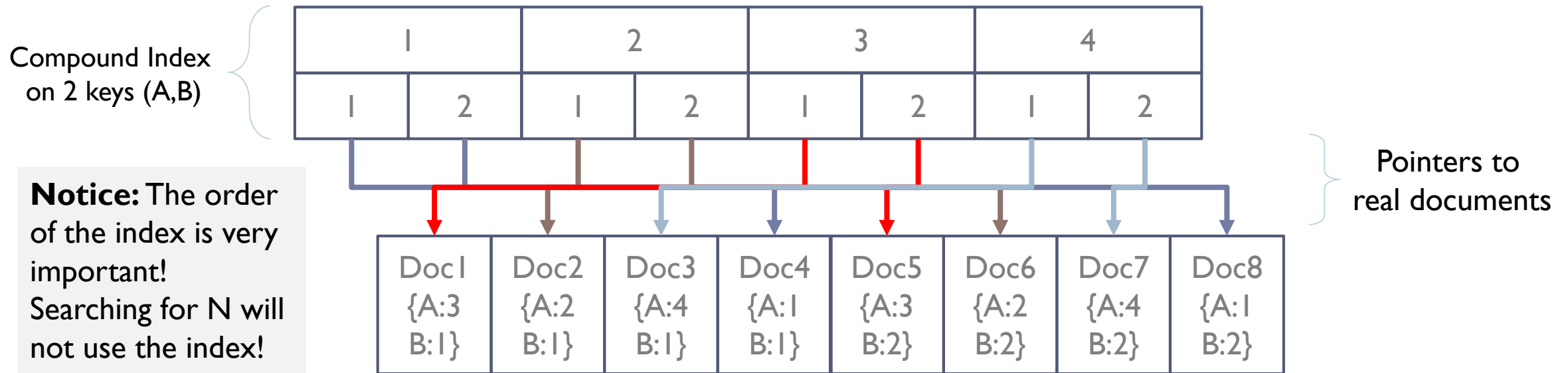
- ▶ creates an ascending index on the score field of the records collection

```
db.records.createIndex( { score: 1 } )
```



Compound Indexes

- ▶ where a single index structure holds references to multiple fields within a collection's documents.



Notice: The index will be updated for every `insert()` operation (slow). Though reading from the index is very fast.

The following index will contain references to documents sorted first by the values of the `item` field and, within each value of the `item` field, sorted by values of the `stock` field.

```
db.products.createIndex( { "item": 1, "stock": 1 } )
```

Index Operations

```
// create a compound index on Key1 ASC, Key2 DESC  
db.colName.createIndex({key1:1, key2:-1})
```

```
// show a list of all indexes in the collection  
db.colName.getIndexes()
```

```
// drop the index Key2 DESC  
db.colName.dropIndex({key2:-1})
```

```
// show useful information about index size  
db.colName.stats()
```

Note: Creating an index will take time and space!

MultiKey Indexes

- ▶ MongoDB supports index for **Array typed keys**. Only one Array typed key is allowed and MongoDB will create a compound key for all value combinations.

```
{
  name: '',
  sports: ['cycling', 'swimming'],
  address: [ {type: 'home', location: ['IA', 'Fairfield']},
             {type: 'work', location: ['NY', 'New York']} ]
}
```

```
db.colName.createIndex({name:1, sports:1})
db.colName.createIndex({name:1, address:1})
db.colName.createIndex({'address.type':1}) // supports rich documents
db.colName.createIndex({'address.location':1})
db.colName.createIndex({sports:1, address:1}) // will fail
```

MultiKey Index & Covered Query

- ▶ A **covered query** is a query that can be satisfied entirely using an index and does not have to examine any documents. An index covers a query when both of the following apply:
 - ▶ all the fields in the query are part of an index, **and**
 - ▶ all the fields returned in the results are in the same index

```
db.col.insert({a:1, b:1})  
db.col.createIndex({a:1, b:1}) // Create Compound Index  
db.col.find({a:1}) // Not Covered  
db.col.find({a:1, b:1, _id:0}).explain() // Covered Index
```

```
db.col.insert({a:[1,2,3], b:1})  
db.col.find({a:1}) // MultiKey Index
```

```
// Will fail: cannot index parallel arrays  
db.col.insert({a:[1,2,3], b:[4,5,6]})
```

Unique Index

```
db.col.insert({a:1, b:1})
```

```
db.col.insert({a:2, b:2})
```

```
// create a unique index on "a"
```

```
db.col.createIndex({a:1},{unique: true})
```

```
db.col.insert({a:2, b:2}) // will fail
```

Sparse Index

- ▶ **Sparse indexes** only contain entries for documents that have the indexed field. The index skips over any document that is missing the indexed field. The index is “sparse” because it does not include all documents of a collection.

```
db.col.insert({a:1, b:1})
```

```
db.col.insert({a:2, b:2})
```

```
db.col.insert({a:3})
```

```
db.col.insert({a:4})
```

```
// create a unique index on "b"
```

```
db.col.createIndex({b:1},{unique: true}) // will fail duplicate "null"
```

```
// create index that doesn't have reference to the last two documents!
```

```
db.col.createIndex({b:1},{unique: true, sparse: true})
```


```
db.col.find({b:null}) // will use BasicCursor and return 2 docs
```

```
db.col.find({b:null}).hint({b:1}) // will use "b" index and return 0 docs
```

Index Selectivity

- ▶ It's important to understand our data before we select an index. Let's say we want to keep a log of activities in our application, choosing an index with more varieties will cut the time of searching and make our operations more efficient.

type	date
save	12 sep 2016
update	12 sep 2016
update	13 sep 2016
save	13 sep 2016
save	13 sep 2016



An index of (type, date) is not a good option as type is limited. While an index of (date, type) will improve the system performance because we can reach to the documents much faster

FullText Index

- ▶ When looking for a string value in MongoDB, the full string will be checked against. While regular expressions could be a solution, it's very slow because it will search the entire collection. FullText Index is the best way to build an index for all words.

```
db.col.insert({mytext:'Hello CS572 Class!'})
db.col.find({mytext:'Hello CS572 Class'}) // will find 0 docs
db.col.find({mytext:'CS572'}) // will find 0 docs
db.col.find({mytext: {$regex: 'CS572'} }) // will find 1 doc but slow!
```

```
db.col.createIndex({mytext:'text'})
// will find our document fast using FullText index. spaces treated as OR
db.col.find({$text: {$search:'Class CS572'} })
```

- ▶ **Note:** MongoDB will log out all slow queries (more than 100ms) automatically to the shell. To write the log on the disk use a [Profiler](#)

Resources

- ▶ Aggregation

- ▶ <https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/>

- ▶ Indexes

- ▶ <https://docs.mongodb.com/manual/indexes/>

Homework 1

1. Continue on yesterday's homework
2. Find below structure for restaurants collection, download [here](#).

```
{
  "address": { "building": "1007",
               "coord": [ -73.856077, 40.848447 ],
               "street": "Morris Park Ave",
               "zipcode": "10462" },
  "district": "Bronx",
  "cuisine": "Bakery",
  "grades": [ {"date": {"$date": 1393804800000}, "grade": "A", "score": 2},
              {"date": {"$date": 1378857600000}, "grade": "A", "score": 6},
              {"date": {"$date": 1358985600000}, "grade": "A", "score": 10},
              {"date": {"$date": 1322006400000}, "grade": "A", "score": 9},
              {"date": {"$date": 1299715200000}, "grade": "B", "score": 14}],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

Questions

1. Write a MongoDB query to display all the documents in the collection restaurants.
2. Write a MongoDB query to display the fields restaurant_id, name, district and cuisine for all the documents in the collection restaurant.
3. Write a MongoDB query to display the fields restaurant_id, name, district and cuisine, but exclude the field _id for all the documents in the collection restaurant.
4. Write a MongoDB query to display the fields restaurant_id, name, district and zipcode, but exclude the field _id for all the documents in the collection restaurant.
5. Write a MongoDB query to display all the restaurant which is in the district Bronx.

Questions

6. Write a MongoDB query to display the first 5 restaurant which is in the district Bronx.
7. Write a MongoDB query to display the next 5 restaurants after skipping first 5 which are in the district Bronx.
8. Write a MongoDB query to find the restaurants which locates in latitude value less than -95.754168.
9. Write a MongoDB query to find the restaurants that does not prepare any cuisine of 'American' and their grade score more than 70 and latitude less than -65.754168.
10. Write a MongoDB query to find the restaurant Id, name, district and cuisine for those restaurants which contains 'Wil' as first three letters for its name.

Questions

11. Write a MongoDB query to find the restaurant Id, name, district and cuisine for those restaurants which contains 'ces' as last three letters for its name.
12. Write a MongoDB query to find the restaurant Id, name, district and cuisine for those restaurants which contains 'Reg' as three letters somewhere in its name.
13. Write a MongoDB query to find the restaurants which belongs to the district Bronx and prepared either American or Chinese dish.
14. Write a MongoDB query to find the restaurant Id, name, district and cuisine for those restaurants which belongs to the district Staten Island or Queens or Bronx or Brooklyn.
15. Write a MongoDB query to find the restaurant Id, name, district and cuisine for those restaurants which are not belonging to the district Staten Island or Queens or Bronx or Brooklyn.

Questions

16. Write a MongoDB query to find the restaurant Id, name, district and cuisine for those restaurants which achieved a score which is not more than 10.
17. Write a MongoDB query to find the restaurant Id, name, address and geographical location for those restaurants where 2nd element of coord array contains a value which is more than 42 and up to 52.
18. Write a MongoDB query to arrange the name of the restaurants in ascending order along with all the columns.
19. Write a MongoDB query to arrange the name of the restaurants in descending order along with all the columns.
20. Write a MongoDB query to arrange the name of the cuisine in ascending order and for those same cuisine district should be in descending order.