

Index & Aggregation

CS415 – Relational and Document-Based Databases

Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

Content

- Index and Explain
- Aggregation framework
 - Grouping with sum, avg
 - Project
 - Match
 - Sort, limit, skip
 - Push and unwind
 - Lookup

Indexes

In any database, indexes support the efficient execution of queries. Without them, the database must scan every document in a collection or table to select those that match the query statement.

Indexes let you query the data in the table using an **alternate key**, in addition to queries against the primary key. Improves querying performance on a non-primary attribute.

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.

Don't overuse indexes! It costs more and consumes resources(memory, disk) as it needs to update the index data structure when writing. At most, have 2 indexes.

GameScores

UserId	GameTitle	TopScore	TopScoreDateTime	Wins	Losses	
"101"	"Galaxy Invaders"	5842	"2015-09-15:17:24:31"	21	72	...
"101"	"Meteor Blasters"	1000	"2015-10-22:23:18:01"	12	3	...
"101"	"Starship X"	24	"2015-08-31:13:14:21"	4	9	...
"102"	"Alien Adventure"	192	"2015-07-12:11:07:56"	32	192	...
"102"	"Galaxy Invaders"	0	"2015-09-18:07:33:42"	0	5	...
"103"	"Attack Ships"	3	"2015-10-19:01:13:24"	1	8	...
"103"	"Galaxy Invaders"	2317	"2015-09-11:06:53:00"	40	3	...
"103"	"Meteor Blasters"	723	"2015-10-19:01:13:24"	22	12	...
"103"	"Starship X"	42	"2015-07-11:06:53:00"	4	19	...
...	

GameTitleIndex

GameTitle	TopScore	UserId
"Alien Adventure"	192	"102"
"Attack Ships"	3	"103"
"Galaxy Invaders"	0	"102"
"Galaxy Invaders"	2317	"103"
"Galaxy Invaders"	5842	"101"
"Meteor Blasters"	723	"103"
"Meteor Blasters"	1000	"101"
"Starship X"	24	"101"
"Starship X"	42	"103"

...

...

...

If you need an item (game score) by GameTitle, you would need to **scan** every single record that is inefficient as your table gets bigger.

To speed up, you can create an index with GameTitle. The table's **primary key attribute** is always projected into an index.

Explain

The explain command provides information on the execution of the following commands: aggregate, count, distinct, find, findAndModify, delete, mapReduce, and update.

```
db.indexC.find().explain("executionStats")
```

Index

`db.collection.getIndexes()` – By default, there is only one index which is the partition key.

`db.collection.createIndex({"columnName": 1})` – Creating an index on a column.

`db.collection.dropIndex("indexName")` – Deleting an index in case you don't need it anymore.

Learn more: <https://www.mongodb.com/docs/manual/indexes/#std-label-index-types>

Aggregation Framework

Aggregation operations process data records and return computed results. Aggregation operations **group values** (*Similar to Group By in SQL*) 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()`)

Aggregation Pipeline

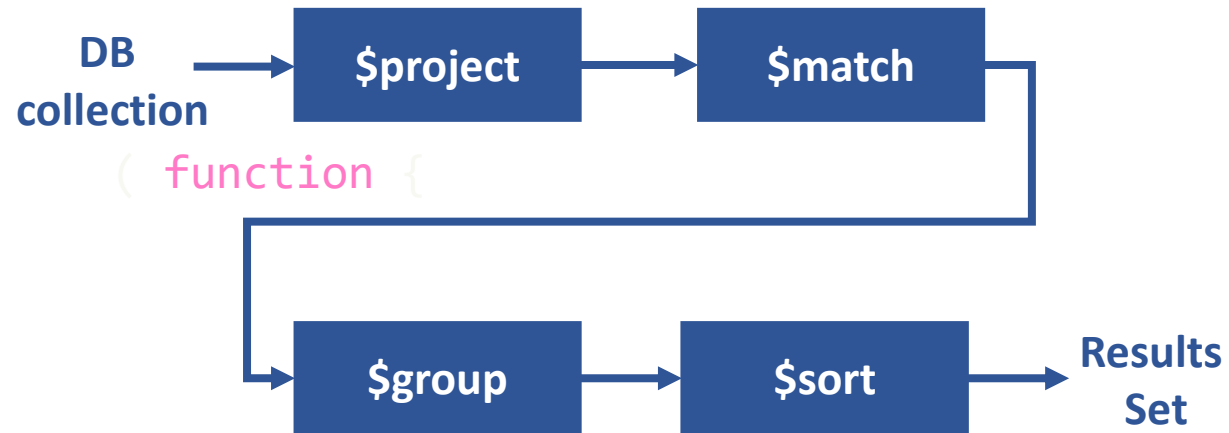
The aggregation framework is built on the concept of **data processing pipelines**. Documents enter a multi-stage pipeline that transforms the documents into an **aggregated result**.

- 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 it's done at the beginning of the aggregation pipeline).
- Every step can appear multiple times in the pipeline.

Aggregation Pipeline Stages

`$group`
`$project`
`$match`
`$sort`
`$limit`
`$skip`
`$unwind`
`$out`
`$lookup`

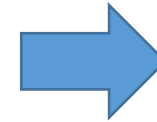
There is a 100 MB limit for any pipeline stage.



Takes the documents returned by the aggregation pipeline and writes them to a specified collection

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

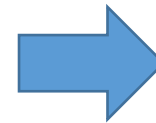


manufacturer	count
Apple	3
Google	1

`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} }  
  ]})
```

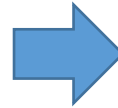


```
{ _id: 'Apple', num_products: 3 }  
{ _id: 'Google', num_products: 1 }
```

Every field must be an accumulator object

\$group

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

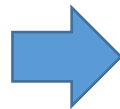


```
{ _id: 'Apple', num_products: 3 }
{ _id: 'Google', num_products: 1 }
```

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



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.



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

Compound Grouping

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



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

Primary Key

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([  
  {$group: { _id: { "maker": "$manufacturer" },  
              sum_prices: {$sum: "$price"} } }  
])
```

```
{ "_id" : { "maker" : "Google" }, "sum_prices" : 500 }  
{ "_id" : { "maker" : "Apple" }, "sum_prices" : 2300 }
```

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

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

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

```
{ "_id" : { "maker" : "Google" }, "maxprice" : 500 }  
{ "_id" : { "maker" : "Apple" }, "maxprice" : 900 }
```

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([  
  {$group: { _id: { "maker": "$manufacturer" },  
             categories: {$addToSet: "$category"} } }  
])
```

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

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

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

Double Grouping

```
{ "_id" : 1, "class_id": 11, "student_id": 123456, "type" : "quiz", "grade" : 95 }
{ "_id" : 2, "class_id": 11, "student_id": 123456, "type" : "assignment", "grade" : 100 }
{ "_id" : 3, "class_id": 11, "student_id": 234456, "type" : "assignment", "grade" : 85 }
{ "_id" : 4, "class_id": 22, "student_id": 123456, "type" : "quiz", "grade" : 90 }
```

We want to find out the **average grade per class**. We assume that there are different number of grades per assignments for each student. The only way to do that is by using **two stages** of `$group` aggregation.

```
db.grades.aggregate([
  {'$group':{_id:{class_id:"$class_id", student_id:"$student_id"},
             'average':{"$avg":"$grade"}}},
  {'$group':{_id:"$_id.class_id",
             'average':{"$avg":"$average"}}}
])
```

1. Calculate the average grade per student in every class.

\$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.zips.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, \$skip and \$limit

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

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

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.

Exercise

Use the aggregation framework to get the largest city in every state using the Zipcode 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.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:

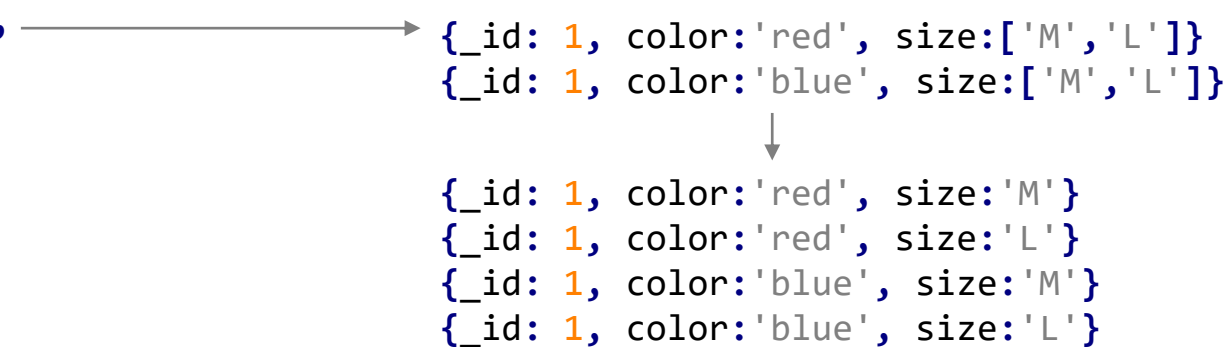
1. Get the population of every city in every state
2. Sort by state, population
3. Group by state, get the first item in each group
4. Now sort by state again (Group might change the order)

\$unwind

Deconstructs an array field from the input documents to output a document for each element. Each output document is the input document with the value of the array field replaced by the element.

```
db.items.drop();
db.items.insert({ _id: 1,
                  'color':['red', 'blue'],
                  'size':['M', 'L'] });
```

```
db.items.aggregate([
  {$unwind:"$color"},
  {$unwind:"$size"}
]);
```



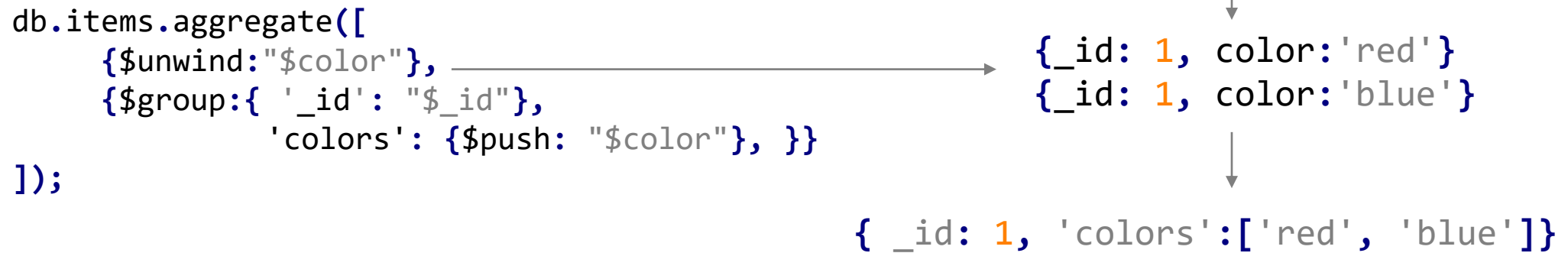
The diagram illustrates the process of unwinding an array field. An arrow points from the aggregation pipeline to a set of two documents. A downward arrow then points from these documents to a final set of four documents, showing how each element of the original array is expanded into its own document.

```
{_id: 1, color:'red', size:['M','L']}
{_id: 1, color:'blue', size:['M','L']}
```

```
{_id: 1, color:'red', size:'M'}
{_id: 1, color:'red', size:'L'}
{_id: 1, color:'blue', size:'M'}
{_id: 1, color:'blue', size:'L'}
```

\$push vs. \$unwind

\$push enables you to reverse the effects of an **\$unwind**



\$lookup

Performs a **left outer join** to an unsharded collection.

```
//orders
{ "_id" : 1, "pk" : "abc", "price" : 12, "quantity" : 2 }
{ "_id" : 2, "pk" : "jkl", "price" : 20, "quantity" : 1 }
//inventory
{ "_id" : 1, "fk" : "abc", "description": "product 1", "instock" : 120 }
{ "_id" : 2, "fk" : "def", "description": "product 2", "instock" : 80 }
{ "_id" : 3, "fk" : "abc", "description": "product 3", "instock" : 60 }
{ "_id" : 4, "fk" : "jkl", "description": "product 4", "instock" : 70 }
```

```
db.orders.aggregate([
  { $lookup: {
    from: "inventory",
    localField: "pk",
    foreignField: "fk",
    as: "inventory_docs" }
  }
])
```



```
{
  "_id" : 1,
  "pk" : "abc",
  "price" : 12,
  "quantity" : 2,
  "inventory_docs" : [ { "_id" : 1, "fk" : "abc",
    "description": "product 1", "instock" : 120 },
    { "_id" : 3, "fk" : "abc", "description": "product 3",
    "instock" : 60 }
  ]
}
...
...
```

SQL to Aggregation Mapping Chart

WHERE	\$match
GROUP BY	\$group
HAVING	\$match
SELECT	\$project
ORDER BY	\$sort
LIMIT	\$limit
SUM()	\$sum
COUNT()	\$sum
join	\$lookup

Count all records from orders using aggregation framework:

SELECT COUNT(*) AS count FROM orders



```
db.orders.aggregate([
  { $group: { _id: null,
              count: { $sum: 1 } } }
])
```


Aggregation Framework Implications

- 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 `$out`, remember that there is 16 MB limit size for every document.
- Aggregation will work fine in sharded environments and the results from `$group` and `$sort` will be sent back to the first sharded DB.