

CC5212-1

PROCESAMIENTO MASIVO DE DATOS

OTOÑO 2021

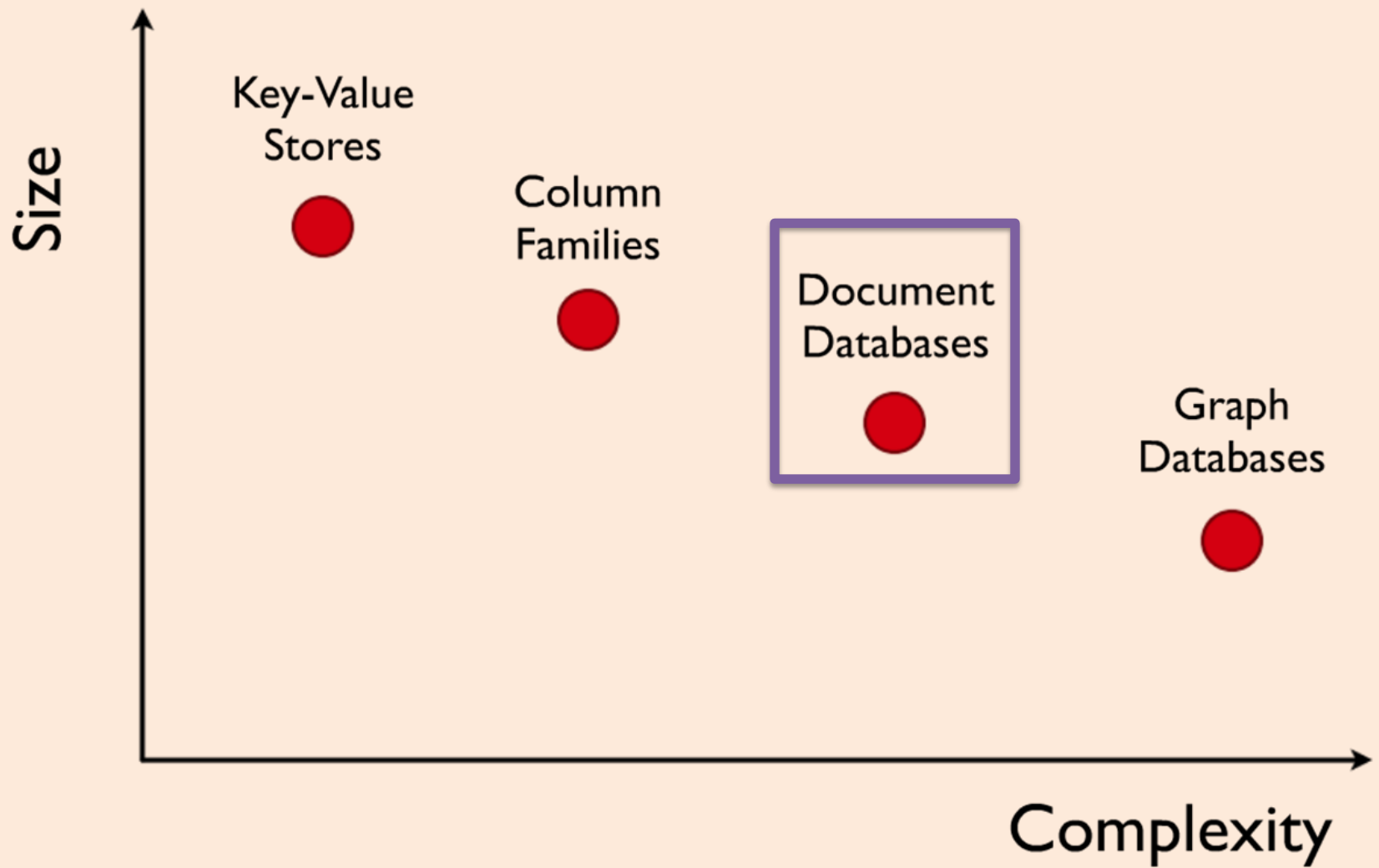
Lecture 10

NoSQL: MongoDB

Aidan Hogan

aidhog@gmail.com

NoSQL



DOCUMENT STORES

Key–Value: a Distributed Map

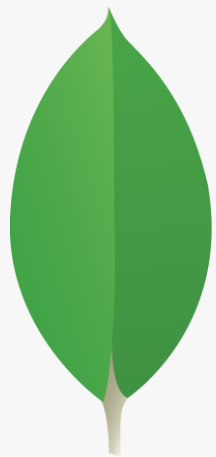
Countries	
Primary Key	Value
Afghanistan	capital:Kabul,continent:Asia,pop:31108077#2011
...	...

Tabular: Multi-dimensional Maps

Countries				
Primary Key	capital	continent	pop-value	pop-year
Afghanistan	Kabul	Asia	31108077	2011
...

Document: Value is a document

Countries	
Primary Key	Value
Afghanistan	{ cap: "Kabul", con: "Asia", pop: { val: 31108077, y: 2011 } }
...	...



mongoDB®

{JSON}

JavaScript Object Notation

Rank			DBMS	Database Model	Score		
May 2021	Apr 2021	May 2020			May 2021	Apr 2021	May 2020
1.	1.	1.	Oracle	Relational, Multi-model	1269.94	-4.98	-75.50
2.	2.	2.	MySQL	Relational, Multi-model	1236.38	+15.69	-46.26
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	992.66	-15.30	-85.64
4.	4.	4.	PostgreSQL	Relational, Multi-model	559.25	+5.73	+44.45
5.	5.	5.	MongoDB	Document, Multi-model	481.01	+11.04	+42.02
6.	6.	6.	IBM Db2	Relational, Multi-model	166.66	+8.88	+4.02
7.	7.	8.	Redis	Key-value, Multi-model	162.17	+6.28	+18.69
8.	8.	7.	Elasticsearch	Search engine, Multi-model	155.35	+3.18	+6.23
9.	9.	9.	SQLite	Relational	126.69	+1.64	+3.66
10.	10.	10.	Microsoft Access	Relational	115.40	-1.33	-4.50
11.	11.	11.	Cassandra	Wide column	110.93	-3.92	-8.22
12.	12.	12.	MariaDB	Relational, Multi-model	96.69	+0.32	+6.61
13.	13.	13.	Splunk	Search engine	92.11	+3.62	+4.36
14.	14.	14.	Hive	Relational	76.19	-2.31	-5.35
15.	15.	23.	Microsoft Azure SQL Database	Relational, Multi-model	70.46	-1.39	+27.70
16.	16.	16.	Amazon DynamoDB	Multi-model	70.07	-0.66	+5.35
17.	17.	15.	Teradata	Relational, Multi-model	69.98	-0.57	-3.91
18.	18.	20.	SAP HANA	Relational, Multi-model	52.75	-0.69	+2.22
19.	20.	21.	Neo4j	Graph	52.23	+1.19	+2.47
20.	21.	18.	Solr	Search engine, Multi-model	51.19	+0.59	-1.39
21.	19.	17.	SAP Adaptive Server	Relational, Multi-model	49.96	-1.70	-4.03
22.	22.	19.	FileMaker	Relational	46.73	+0.32	-4.23
23.	23.	22.	HBase	Wide column	43.24	-0.92	-6.48
24.	24.	26.	Google BigQuery	Relational	37.63	+2.05	+10.04
25.	25.	24.	Microsoft Azure Cosmos DB	Multi-model	34.71	+1.19	+4.03

MONGODB:

DATA MODEL

JavaScript Object Notation: JSON

```
{
  "id": 179,
  "name": "The Wire",
  "type": "Scripted",
  "language": "English",
  "genres": [ "Drama", "Crime", "Thriller" ],
  "status": "Ended",
  "runtime": 60,
  "premiered": "2002-06-02",
  "schedule": {
    "time": "21:00",
    "days": [
      "Sunday"
    ]
  },
  "rating": {
    "average": 9.4
  }
}
```



Binary JSON: BSON

```
{
  "_id": ObjectId(99a88b77c66d),
  "name": "The Wire",
  "type": "Scripted",
  "language": "English",
  "genres": [ "Drama", "Crime", "Thriller" ],
  "status": "Ended",
  "runtime": 60,
  "premiered": ISODate("2002-06-02"),
  "schedule": {
    "time": "21:00",
    "days": [
      "Sunday"
    ]
  },
  "rating": {
    "average": 9.4
  }
}
```

BSON { 01010100
11101011
10101110
01010101 }

MongoDB: Datatypes

String: "sí"

Object: {}

Array: []

Boolean: true

Double: 43.2

{JSON}
JavaScript Object Notation

etc.

ObjectID: ObjectId(0A0A0A0A0A0A)

Date: ISODate("2003-14-15T09:26:53.589Z")

BSON {
01010100
11101011
10101110
01010101
}

etc.

MongoDB: Map from keys to BSON values

TVSeries	
Key	BSON Value
99a88b77c66d	<pre>{ "_id": ObjectId(99a88b77c66d), "name": "The Wire", "type": "Scripted", "language": "English", "genres": ["Drama", "Crime", "Thriller"], "status": "Ended", "runtime": 60, "premiered": ISODate("2002-06-02"), "schedule": { "time": "21:00", "days": ["Sunday"] }, "rating": { "average": 9.4 } }</pre>
...	...

MongoDB Collection: Similar Documents

TVSeries	
Key	BSON Value
99a88b77c66d	<pre>{ "_id": ObjectId(99a88b77c66d), "name": "The Wire", "type": "Scripted", ... }</pre>
11f22e33d44c	<pre>{ "_id": ObjectId(11f22e33d44c), "name": "Rick and Morty", "type": "Animation", ... }</pre>

MongoDB Database: Related Collections

TVSeries

Key	BSON Value
...	...

TVEpisodes

Key	BSON Value
...	...

TVNetworks

Key	BSON Value
...	...

database: TV

Load database tvdb (and create if not exists):

```
> use tvdb
```

```
switched to db tvdb
```

See all (non-empty) databases (tvdb is empty):

```
> show dbs
```

```
local      0.00001 GB  
test       0.00231 GB
```

See current database:

```
> db
```

```
tvdb
```

Drop current database:

```
> db.dropDatabase()
```

```
{ "dropped" : "tvdb", "ok" : 1 }
```

Create collection series:

```
> db.createCollection("series")  
  
{ "ok" : 1 }
```

See all collections in current database:

```
> show collections  
  
series
```

Drop collection series:

```
> db.series.drop()  
  
true
```

Clear all documents from collection series:

```
> db.series.remove( {} )  
  
WriteResult({ "nRemoved" : 0 })
```

Create capped collection (keeps only most recent):

```
> db.createCollection("last100episodes",  
  { capped: true, size: 12285600, max: 100 } )  
  
{ "ok" : 1 }
```


MONGODB: INSERTING DATA

Insert document: without `_id`

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60,  
    "genres": [  
      "Science-Fiction",  
      "Thriller"  
    ]  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

Insert document: with `_id`

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60,  
    "genres": [  
      "Science-Fiction",  
      "Thriller"  
    ]  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

... fails if `_id` already exists

... use `update` or `save` to update existing document(s)

Use save and `_id` to overwrite:

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

```
> db.series.save(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror (Overwritten)"  
  })
```

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

... overwrites old document

MONGODB: QUERIES WITH SELECTION

Query documents in a collection with `find`:

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60  
  })
```

```
> db.series.find()  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "type" : "Scripted", "runtime" : 60 }
```

... use `findOne()` to return one document

Pretty print results with pretty:

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60  
  })
```

```
> db.series.find().pretty()  
  
{  
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),  
  "name" : "Black Mirror",  
  "type" : "Scripted",  
  "runtime" : 60  
}
```

Selection: find documents matching σ

```
> db.series.find( $\sigma$ )
```


Selection σ : Equality

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60  
  })
```

Equality: { key: value }

```
> db.series.find( { "type": "Scripted" } )  
{ "name" : "Black Mirror", "type" : "Scripted", "runtime" : 60 }
```

Results would include `_id` but for brevity, we will omit this from examples where it is not important.



Selection σ : Nested key

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4 },  
    "runtime": 60  
  })
```

Key can access nested values:

```
> db.series.find( { "rating.avg": 9.4 } )  
  
{ "name" : "Black Mirror", "rating": { "avg": 9.4 }, "runtime" : 60 }
```

Selection σ : Equality on null

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": null },  
    "runtime": 60  
  })
```

Equality on nested null value:

```
> db.series.find( { "rating.avg": null } )  
  
{ "name" : "Black Mirror", "rating": { "avg": null }, "runtime" : 60 }
```

... matches when value is null or ...

Selection σ : Equality on null

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "val": 9.4 },  
    "runtime": 60  
  })
```

Key can access nested values:

```
> db.series.find( { "rating.avg": null } )  
  
{ "name" : "Black Mirror", "rating": { "val": 9.4 }, "runtime" : 60 }
```

... when field doesn't exist with key.

Selection σ : Equality on document

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4 },  
    "runtime": 60  
  })
```

Value can be an object/document:

```
> db.series.find( { "rating": { "avg": 9.4 } } )  
{ "name" : "Black Mirror", "rating": { "avg": 9.4 }, "runtime" : 60 }
```

Selection σ : Equality on document

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4, "votes": 9001 },  
    "runtime": 60  
  })
```

Value can be an object/document:

```
> db.series.find( { "rating": { "votes":9001 } } )
```

... no results: needs to match full object.

Selection σ : Equality on document

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4, "votes": 9001 },  
    "runtime": 60  
  })
```

Value can be an object/document:

```
> db.series.find( { "rating": { "votes":9001, "avg":9.4 } } )
```

... no results: order of attributes matters.

Selection σ : Equality on exact array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Equality can match an exact array:

```
> db.series.find( { "genres": [ "Science-Fiction",  
                               "Thriller" ] } )  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```


Selection σ : Equality on exact array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Equality can match an exact array

```
> db.series.find( { "genres": [ "Science-Fiction" ] } )
```

... no results: needs to match full array.

Selection σ : Equality on exact array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Equality can match an exact array

```
> db.series.find( { "genres": [ "Thriller",  
                               "Science-Fiction" ] } )
```

... no results: order of elements matters.

Selection σ : Equality matches inside array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Equality matches a value in an array:

```
> db.series.find( { "genres": "Thriller" } )  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

Selection σ : Equality matches both

```
> db.series.insert( { "name": "A" , "val": [ 5, 6 ] } )  
> db.series.insert( { "name": "B" , "val": 5 } )
```

Equality matches a value inside and outside an array:

```
> db.series.find( { "val": 5 } )  
  
{ "name": "A" , "val": [ 5, 6 ] }  
{ "name": "B" , "val": 5 }
```

cough

Selection σ : Inequalities

Less than: `{ key: { $lt: value } }`

Greater than: `{ key: { $gt: value } }`

Less than or equal: `{ key: { $lte: value } }`

Greater than or equal: `{ key: { $gte: value } }`

Not equals: `{ key: { $ne: value } }`

Selection σ : Less Than

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Less than: { key: { \$lt: value } }

```
> db.series.find({ "runtime": { $lt: 70 } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

Selection σ : Match one of multiple values

Match any value: { key: { \$in: [v1, ..., vn] } }

Match no value: { key: { \$nin: [v1, ..., vn] } }

... also passes if key does not exist.

Selection σ : Match one of multiple values

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Match any value: { key: { \$in: [v1, ..., vn] } }

```
> db.series.find({ "runtime": { $in: [30, 60] } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```


Selection σ : Match one of multiple values

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Match any value: { key: { \$in: [v1, ..., vn] } }

```
> db.series.find({ "genres": { $in: ["Noir", "Thriller"] } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

... if key references an array, any value of array should match any value of \$in

Selection σ : Boolean connectives

And: $\{ \$and: [\sigma , \sigma'] \}$

Or: $\{ \$or: [\sigma , \sigma'] \}$

Not: $\{ \$not: [\sigma] \}$

Nor: $\{ \$nor: [\sigma , \sigma'] \}$

... naturally, we can nest such conditions

Selection σ : And

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

And: $\{ \$and: [\sigma , \sigma'] \}$

```
> db.series.find({ $and: [  
  { "runtime": { $in: [30, 60] } } ,  
  { "name": { $ne: "Lost" } } ] }  
)  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

Selection σ : Attribute (not) exists

Exists: `{ key: { $exists : true } }`

Not Exists: `{ key: { $exists : false } }`

Selection σ : Attribute exists

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Exists: { key: { \$exists : true } }

```
> db.series.find({ "name": { $exists : true } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

... checks that the field key exists
(even if value is NULL)

Selection σ : Attribute not exists

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Not exists: { key: { \$exists : false } }

```
> db.series.find({ "name": { $exists : false } })
```

... checks that the field key doesn't exist
(empty results)

Selection σ : Arrays

All: `{ key: { $all : [v1, ..., vn] } }`

Match one: `{ key: { $elemMatch : { σ 1, ..., σ n } } }`

Size: `{ key: { $size : int } }`

Selection σ : Array contains (at least) all elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Sci-Fi", "Thriller", "Comedy" ],  
    "runtime": 60  
  })
```

All: `{ key: { $all : [v1, ..., vn] } }`

```
> db.series.find(  
  { "genres": { $all : [ "Comedy", "Sci-Fi" ] } })  
  
{ "name" : "Black Mirror", "genres": [ "Sci-Fi", "Thriller", "Comedy" ],  
  "runtime" : 60 }
```

... all values are in the array

Selection σ : Array element matches (with AND)

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Match one: { key: { \$elemMatch : { σ_1 , ..., σ_n } } }

```
> db.series.find(  
  { "series": { $elemMatch : { $gt: 1, $lt: 3 } } })  
  
{ "name" : "Black Mirror", "series": [ 1, 2, 3 ], "runtime" : 60 }
```

... one element matches all criteria (with AND)

Selection σ : Array with exact size

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Size: `{ key: { $size : int } }`

```
> db.series.find( { "series": { $size : 3 } })  
{ "name" : "Black Mirror", "series": [ 1, 2, 3 ], "runtime" : 60 }
```

... only possible for exact size of array (not ranges)

Selection σ : Type of value

Type: `{ key: { $type: typename } }`

`"timestamp"` `"decimal"`
`"string"`
`"double"` `"binData"`
`"date"` `"object"` `"int"`
`"array"` `"long"`
`"bool"` `"undefined"`
`"objectId"` `"null"` `"regex"`
`"dbPointer"` `"array"`
`"javascript"` `"number"`
...

Selection σ : Type of value

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Type: { key: { \$type: typename } }

```
> db.series.find({ "runtime": { $type : "number" } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

Selection σ : Matching an array by type?

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Type: { key: { \$type: typename } }

```
> db.series.find({ "genres": { $type : "array" } })
```

... empty
... passes if any value in the array has that type

Selection σ : Matching an array by type?

Arrays

When applied to arrays, `$type` matches any inner element that is of the specified **BSON** type. For example, when matching for `$type : 'array'`, the document will match if the field has a nested array. It will not return results where the field itself is an array.

<https://docs.mongodb.com/manual/reference/operator/query/type/>

**BUT WHAT IF
I WANT TO CHECK**



**IF THE VALUE
IS AN ARRAY?**

imgflip.com

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

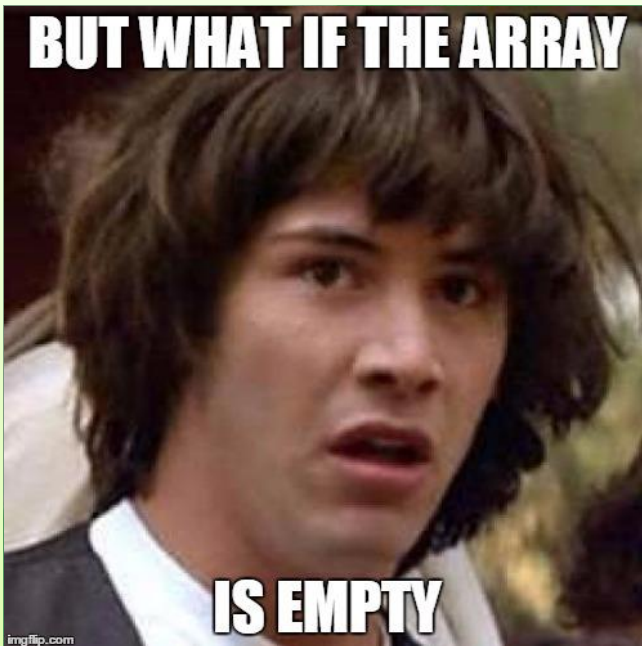
```
> db.series.find(  
  { "genres": { $elemMatch: { $exists : true } } }  
)  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller"  
], "runtime" : 60 }
```

Selection σ : Matching an array by type?

Arrays

When applied to arrays, `$type` matches any **inner** element that is of the specified **BSON** type. For example, when matching for `$type : 'array'`, the document will match if the field has a nested array. It will not return results where the field itself is an array.

<https://docs.mongodb.com/manual/reference/operator/query/type/>



```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [],  
    "runtime": 60  
  })
```

```
> db.series.find(  
  { $or: [  
    { "genres": { $elemMatch: { $exists: true } } },  
    { "genres": [] }  
  ] } )
```

```
{ "name" : "Black Mirror", "genres": [], "runtime" : 60 }
```

Selection σ : Other operators

Mod: `{ key: { $mod [div, rem] } }`

Regex: `{ key: { $regex: pattern } }`

Text search: `{ $text: { $search: terms } }`

Where (JS): `{ $where: javascript_code }`

... where is executed over all documents
(and should be avoided where possible)

Selection σ : Geographic features



Selection σ : Bitwise features

`$bitsAllClear`

`$bitsAllSet`

`$bitsAnyClear`

`$bitsAnySet`

...

<https://docs.mongodb.com/manual/reference/operator/query-bitwise/>

MONGODB:

PROJECTION OF OUTPUT VALUES

Projection π : Choose output values

<code>key: 1:</code>	Output field(s) with <code>key</code>
<code>key: 0:</code>	Suppress field(s) with <code>key</code>
<code>array.\$: 1</code>	Project first matching array element I
<code>\$elemMatch:</code>	Project first matching array element II
<code>\$slice:</code>	Output first or last <code>slice</code> array values

Projection π : Output certain fields

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Project only certain fields: $\{ k1: 1, \dots, kn: 1 \}$

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "name": 1 , "runtime": 1, "network": 1 })  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "runtime" : 60 }
```

... outputs what is available; by default also outputs `_id` field

Projection π : Output embedded fields

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4 , "votes": 9001 },  
    "runtime": 60  
  })
```

Project (embedded) fields: $\{ k_1: 1, \dots, k_n: 1 \}$

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "name": 1 , "rating.avg": 1 })  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "rating" : { "avg": 9.4 } }
```

... field is still nested in output

Projection π : Output embedded fields in array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "reviews": [  
      { "user": "jack" , "score": 9.1 },  
      { "user": "jill" , "score": 8.3 }  
    ],  
    "runtime": 60  
  })
```

Project (embedded) fields: $\{ k1: 1, \dots, kn: 1 \}$

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "name": 1 , "reviews.score": 1 })  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "reviews" : [ { "score": 9.1 } , { "score": 8.3 } ] }
```

... projects from within the array.

Projection π : Suppress certain fields

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Return all but specified fields: $\{ k_1: 0, \dots, k_n: 0 \}$

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "series": 0 })  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "runtime" : 60 }
```

... cannot combine 0 and 1 except ...

Projection π : Suppress certain fields

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Suppress ID: `{ _id: 0, k1: 1, ..., kn: 1 }`

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "_id": 0, "name": 1, "series": 1 })  
  
{ "name" : "Black Mirror", "series" : [ 1, 2, 3 ] }
```

... 0 suppresses `_id` when other fields are output

Projection π : Output first matching element

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Project first matching element: `array.$: 1`

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series.$": 1 } )  
  
{ _id: ..., "series": [ 2 ] }
```

Projection π : Output first matching element

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Project first matching element: $\$elemMatch$

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $elemMatch: { $lt: 3 } } } )  
  
{ "_id": ..., "series": [ 1 ] }
```

... allows to separate selection and projection criteria.

Projection π : Output first matching element

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Project first matching element: $\$elemMatch$

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $elemMatch: { $gt: 3 } } } )  
  
{ "_id": ... }
```

... drops array field entirely if no element is projected.

Projection π : Output first matching element

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "reviews": [  
      { "user": "jack" , "score": 9.1 },  
      { "user": "jill" , "score": 8.3 }  
    ]  
  }  
)
```

Project first matching element: `$elemMatch`

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "reviews": { $elemMatch: { "score": { $gt: 8 } } } } )  
  
{ "_id": ..., "reviews": [ { "user": "jack" , "score": 9.1 } ] }
```

... can match on array of documents.

Projection π : Output some matching elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Return first n elements: $\$slice: n$ (where $n > 0$)

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $slice: 2 } } )  
  
{ "_id": ..., "name": "Black Mirror", "series": [ 1, 2 ], "runtime": 60 }
```

Projection π : Output some matching elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Return last n elements: $\$slice: n$ (where $n < 0$)

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $slice: -2 } } )  
  
{ "_id": ..., "name": "Black Mirror", "series": [ 2, 3 ], "runtime": 60 }
```

Projection π : Output some matching elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Skip n and return m: `$slice: [n,m]` ($n, m > 0$)

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $slice: [ 2, 1 ] } } )  
  
{ "_id": ..., "name": "Black Mirror", "series": [ 3 ], "runtime": 60 }
```


Projection π : Output some matching elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

From last n , return m : $\$slice: [n,m]$ ($n < 0, m > 0$)

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $slice: [ -2, 1 ] } } )  
  
{ "_id": ..., "name": "Black Mirror", "series": [ 2 ], "runtime": 60 }
```

MONGODB:

UPDATES

Update **u**: Modify fields in documents

\$set :	Set the value
\$unset :	Remove the key and value
\$rename :	Rename the field (change the key)
\$setOnInsert :	Set value only if a new document is created
\$inc :	Increment number by inc
\$mul :	Multiply number by mul
\$min :	Replace values less than min by min
\$max :	Replace values greater than max by max
\$currentDate :	Set to current date

Use update with **query** criteria and **update** criteria:

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "type": "Scripted",
    "language": "English",
  })

WriteResult({ "nInserted" : 1 })

> db.series.update(
  { "type": "Scripted" },
  { $set: { "type": "Fiction" } },
  { "multi": true }
)

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.series.find({ "name": "Black Mirror" })

{
  "_id": ObjectId("5951e0b265ad257d48f4a7d5"),
  "name": "Black Mirror",
  "type": "Fiction",
  "language": "English"
}
```

Update **u**: Modify arrays in documents

- \$addToSet**: Adds value if not already present
- \$pop**: Deletes first or last value
- \$push**: Appends (an) item(s) to the array
- \$pullAll**: Removes values from a list
- \$pull**: Removes values that match a condition

(Sub-)operators used upon pushing/adding values:

- \$**: Select first element matching query condition
- \$each**: Add or push multiple values
- \$slice**: After pushing, keep first or last **slice** values
- \$sort**: Sort the array after pushing [1 for ASC; -1 for DESC]
- \$position**: Push values to a specific array index

Update **u**: Modify arrays in documents

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "type": [ "Scripted", "Drama" ],
    "languages": [ "English", "Spanish" ],
  })

WriteResult({ "nInserted" : 1 })

> db.series.update(
  { "type": "Scripted" },
  { $pull: { "type": { $in: [ "Drama", "Sci-Fi" ] } },
    "languages": "Spanish" } },
  { "multi": true } )

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.series.find({ "name": "Black Mirror" })

{
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),
  "name" : "Black Mirror",
  "type" : [ "Scripted" ],
  "languages" : [ "English" ] }
```

Update **u**: Modify arrays in documents

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "ratings": [ 8, 11, 13, 9 ],  
    "languages": [ "English", "Spanish" ],  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

```
> db.series.update(  
  { "ratings": { $gt: 10 } },  
  { $set: { "ratings.$": 10 } },  
  { "multi": true } )
```

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

```
> db.series.find({ "name": "Black Mirror" })  
  
{  
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),  
  "name" : "Black Mirror",  
  "ratings" : [ 8, 10, 13, 9 ],  
  "languages" : [ "English", "Spanish" ] }
```

Update **u**: Modify arrays in documents

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "ratings": [ 8, 11, 13, 9 ],
    "languages": [ "English", "Spanish" ],
  })

WriteResult({ "nInserted" : 1 })

> db.series.update(
  { "ratings": { $gt: 10 } },
  { $push: { "ratings": { $each: [4, 9],
                                $sort: 1, $slice: 4 } } },
  { "multi": true } )

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.series.find({ "name": "Black Mirror" })

{
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),
  "name" : "Black Mirror",
  "ratings" : [ 4, 8, 9, 9 ],
  "languages" : [ "English", "Spanish" ] }
```


MONGODB:

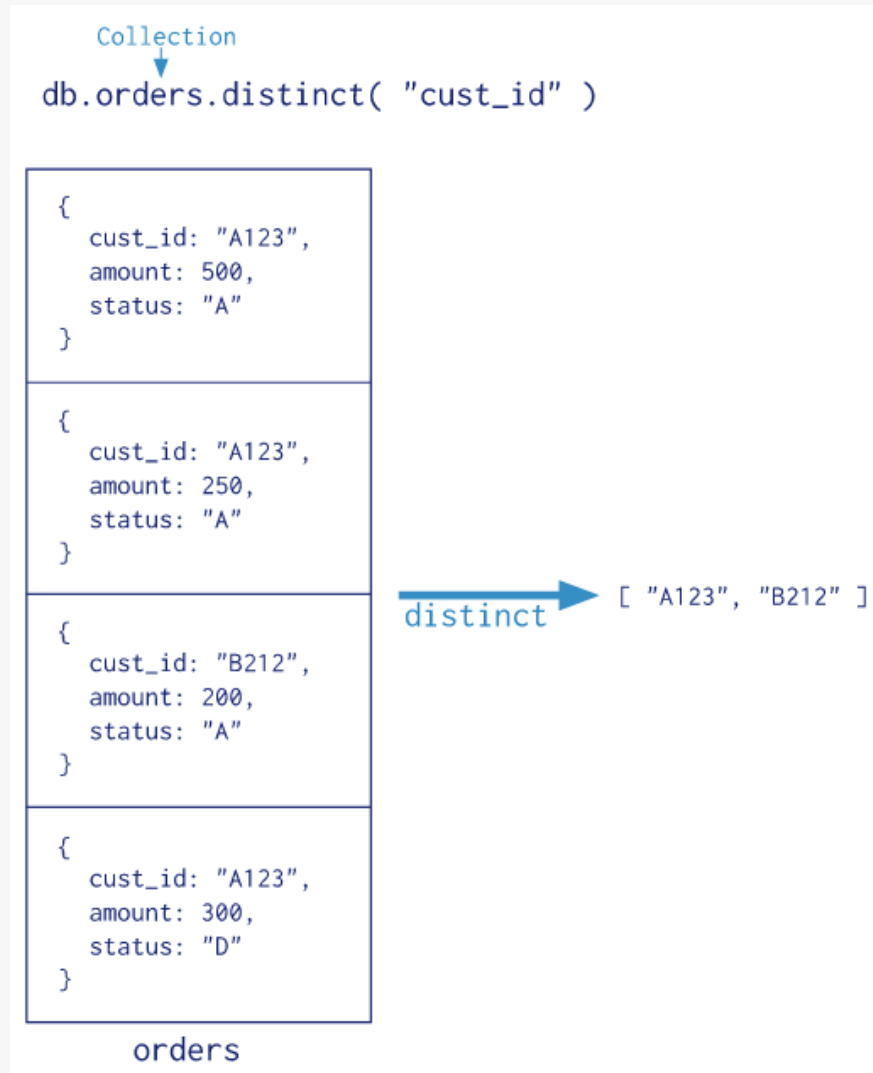
AGGREGATION AND PIPELINES

Aggregation: without grouping

`db.coll.distinct(key)`: Array of unique values for that key
<https://docs.mongodb.com/manual/reference/method/db.collection.distinct/>

`db.coll.count()`: Count the documents
<https://docs.mongodb.com/manual/reference/method/db.collection.count/>

Aggregation: distinct



Pipelines: Transforming Collections



Pipeline ρ : Stage Operators

\$match:	Filter by selection criteria σ
\$project:	Perform a projection π
\$group:	Group documents by a key/value [used with <i>\$sum</i> , <i>\$avg</i> , <i>\$first</i> , <i>\$last</i> , <i>\$max</i> , <i>\$min</i> , etc.]
\$lookup:	Perform left-outer-join with another collection
\$unwind:	Copy each document for each array value
\$collStats:	Get statistics about collection
\$count:	Count the documents in the collection
\$sort:	Sort documents by a given key (ASC DESC)
\$limit:	Return (up to) n first documents
\$sample:	Return (up to) n sampled documents
\$skip:	Skip n documents
\$out:	Save collection to MongoDB

(more besides)

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

Pipeline Aggregation: `$match` and `$group`

Collection
↓
`db.orders.aggregate([`
 `$match stage → { $match: { status: "A" } },`
 `$group stage → { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }`
 `]`)

{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }

orders

`$match` →

{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }

`$group` →

Results	
{ _id: "A123", total: 750 }	
{ _id: "B212", total: 200 }	

Pipeline Aggregation: \$lookup

```
{
  "name": "The Wire",
  "country": "US"
}
{
  "name": "Black Mirror",
  "country": "UK"
}
```

```
{
  "nation": "US",
  "network": "HBO"
}
{
  "nation": "US",
  "network": "FOX"
}
```

```
> db.series.aggregate( [
  { $lookup: { from: "networks", localField: "country",
    foreignField: "nation", as: "possibleNetworks" }
  } ] )
```

```
{
  "name": "The Wire",
  "country": "US",
  "possibleNetworks": [
    { "nation": "US", "network": "HBO" },
    { "nation": "US", "network": "FOX" }
  ]
}
{
  "name": "Black Mirror",
  "country": "UK",
  "possibleNetworks": []
}
```

Pipeline Aggregation: \$unwind

```
{  
  "name": "The Wire",  
  "genres": [ "Drama", "Crime", "Thriller" ]  
}
```

```
> db.series.aggregate( [ { $unwind : "$genres" } ] )
```

```
{  
  "name": "The Wire",  
  "genres": "Drama"  
}  
{  
  "name": "The Wire",  
  "genres": "Crime"  
}  
{  
  "name": "The Wire",  
  "genres": "Thriller"  
}
```


MONGODB:

INDEXING

MongoDB Indexing

Per collection:

- `_id` Index
- Single-field Index (sorted)
- Compound Index (sorted)
- Multikey Index (for arrays)
- Geospatial Index
- Full-text Index
- Hash-based indexing (hashed)

MongoDB: Text Indexing Example

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "summary": "A British sci-fi series with a dystopian setting",  
    "languages": [ "English", "Spanish" ],  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

```
> db.series.createIndex( { summary: "text"})
```

```
> db.series.getIndexes()
```

```
[  
  {  
    "v" : 2,  
    "key" : { "_id" : 1 },  
    "name" : "_id_",  
    "ns" : "test.series"      },  
  {  
    "v" : 2,  
    "key" : { "fts" : "text", "_ftsx" : 1 },  
    "name" : "summary_text",  
    "ns" : "test.series",  
    "weights" : { "summary" : 1 },  
    "default_language" : "english",  
    "language_override" : "language",  
    "textIndexVersion" : 3      }  
]
```

MongoDB: Text Indexing Example

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "summary": "A British sci-fi series with a dystopian setting",
    "languages": [ "English", "Spanish" ],
  })

WriteResult({ "nInserted" : 1 })

> db.series.createIndex( { summary: "text" })

> db.series.find(
  { $text: { $search: "dystopia sci-fi" } } )

{
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),
  "name" : "Black Mirror",
  "summary" : "A British sci-fi series with a dystopian setting",
  "languages" : [
    "English",
    "Spanish"
  ]
}
```

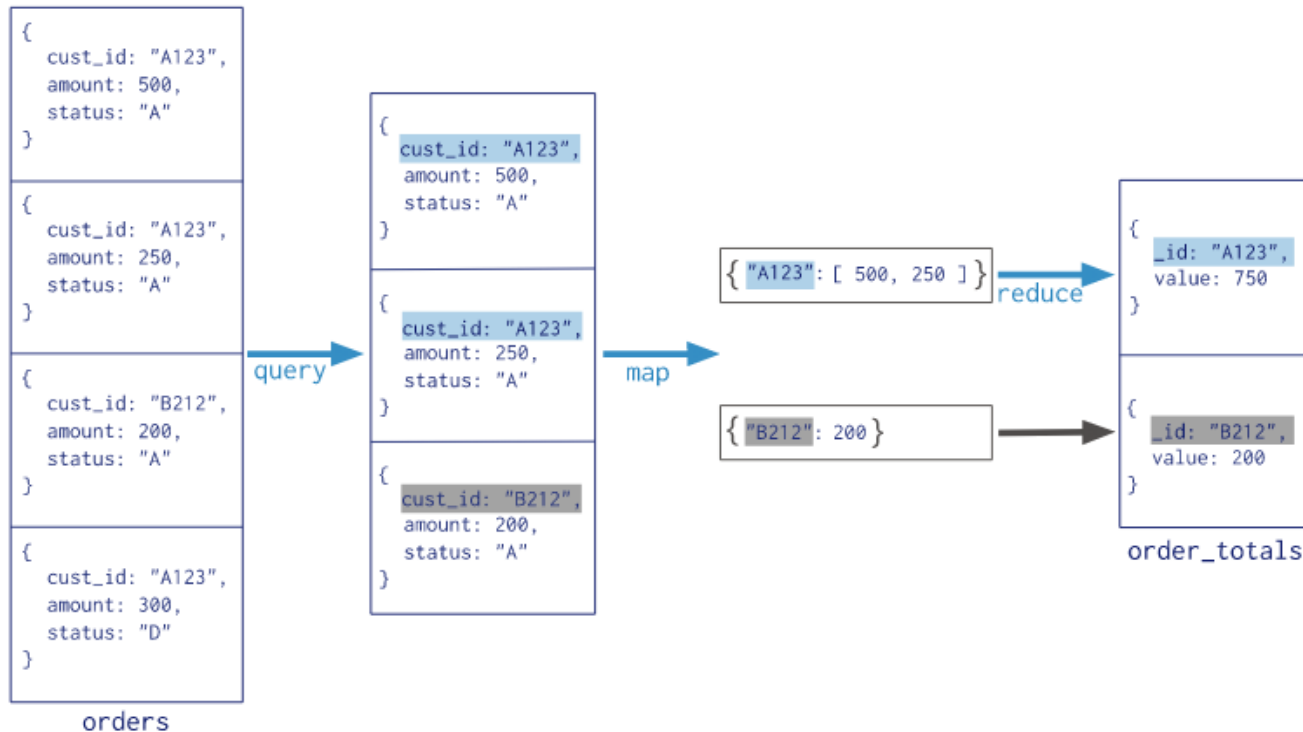
MONGODB: DISTRIBUTION

MongoDB: Distribution

- "Sharding":
 - Hash-based or Horizontal Ranged
(Depends on indexes)
- Replication
 - Replica sets

mapreduce

```
Collection
↓
db.orders.mapReduce(
  map   → function() { emit( this.cust_id, this.amount ); },
  reduce → function(key, values) { return Array.sum( values ) },
  query → {
    query: { status: "A" },
    out: "order_totals"
  }
)
```



MONGODB:

A WORD OF WARNING

Why you should never, ever, ever use MongoDB

19 Jul 2015

MongoDB is evil. It...

- ... loses data (sources: [1](#), [2](#))
- ... in fact, for a long time, ignored errors by default and assumed every single write succeeded no matter what (which on 32-bits systems led to losing all data silently after some 3GB, due to MongoDB limitations)
- ... is slow, *even* at its advertised usecases, and claims to the contrary are completely lacking evidence (sources: [3](#), [4](#))
- ... forces the poor habit of implicit schemas in nearly all usecases (sources: [4](#))
- ... has locking issues (sources: [4](#))
- ... has an atrociously poor response time to security issues - it took them **two years** to patch an insecure default configuration that would expose *all of your data* to anybody who asked, without authentication (sources: [5](#))
- ... is not ACID-compliant (sources: [6](#))
- ... is a nightmare to scale and maintain
- ... isn't even exclusive in its offering of JSON-based storage; PostgreSQL does it too, and other (better) document stores like CouchDB have been around for a long time (sources: [7](#), [8](#))

<http://crypto.net/~joepie91/blog/2015/07/19/why-you-should-never-ever-ever-use-mongodb/>

Ransomware groups have deleted over 10,000 MongoDB databases

Five groups of attackers are competing to delete as many publicly accessible MongoDB databases as possible



By Lucian Constantin

CSO Senior Writer, IDG News Service | JAN 6, 2017 10:02 AM PST

Gerd Altmann (CC0)

Groups of attackers have adopted a new tactic that involves deleting publicly exposed MongoDB databases and asking for money to restore them. In a matter of days, the number of affected databases has risen from hundreds to more than 10,000.

The issue of misconfigured MongoDB installations, allowing anyone on the internet to access sensitive data, is not new. Researchers have been finding such open databases for years, and [the latest estimate](#) puts their number at more than 99,000.

<https://www.computerworld.com/article/3155260/ransomware-groups-have-deleted-over-10000-mongodb-databases.html>

MongoDB "open-source" Server Side Public License rejected

Red Hat won't use MongoDB in Red Hat Enterprise Linux or Fedora thanks to MongoDB's new Server Side Public License.



By [Steven J. Vaughan-Nichols](#) for [Linux and Open Source |](#)

January 16, 2019 -- 19:33 GMT (11:33 PST) | Topic: [Cloud](#)

[MongoDB](#) is an open-source document NoSQL database with a problem. While very popular, cloud companies, such as [Amazon Web Services \(AWS\)](#), [IBM Cloud](#), [Scalegrid](#), and [ObjectRocket](#) has profited from it by offering it as a service while [MongoDB Inc.](#) hasn't been able to monetize it to the same degree. MongoDB's answer? Relicense the program under its new [Server Side Public License \(SSPL\)](#). Open-source powerhouse [Red Hat's](#) reaction? [Drop MongoDB from Red Hat Enterprise Linux \(RHEL\) 8.](#)

Red Hat's Technical and Community Outreach Program Manager Tom Callaway explained, in a note stating MongoDB is being removed from [Fedora Linux](#), that "It is the belief of Fedora that the [SSPL](#) is intentionally crafted to be aggressively discriminatory towards a specific class of users." [Debian Linux](#) had already dropped MongoDB from its distribution.

AI, ML & DATA ENGINEERING

Jepsen Disputes MongoDB's Data Consistency Claims

👍 LIKE

🗨️ 5



MAY 22, 2020 • 4 MIN READ

by

Jonathan Allen

FOLLOW

In an article titled [MongoDB and Jepsen](#), MongoDB claimed that their database passed “the industry’s toughest data safety, correctness, and consistency Tests”. In response, Jepsen published an article stating that MongoDB 3.6.4 had in fact failed their tests; the newer [MongoDB 4.2.6 has more problems](#) including “retrocausal transactions” where a transaction reverses order so that a read can see the result of a future write.

Jepsen LLC’s response begins with this [reply to Maxime Beugnet](#) on their official Twitter feed:

“

I have to admit raising an eyebrow when I saw that web page. In that report, MongoDB lost data and violated causal by default. Somehow that became "among the strongest data consistency, correctness, and safety guarantees of any database available today"!

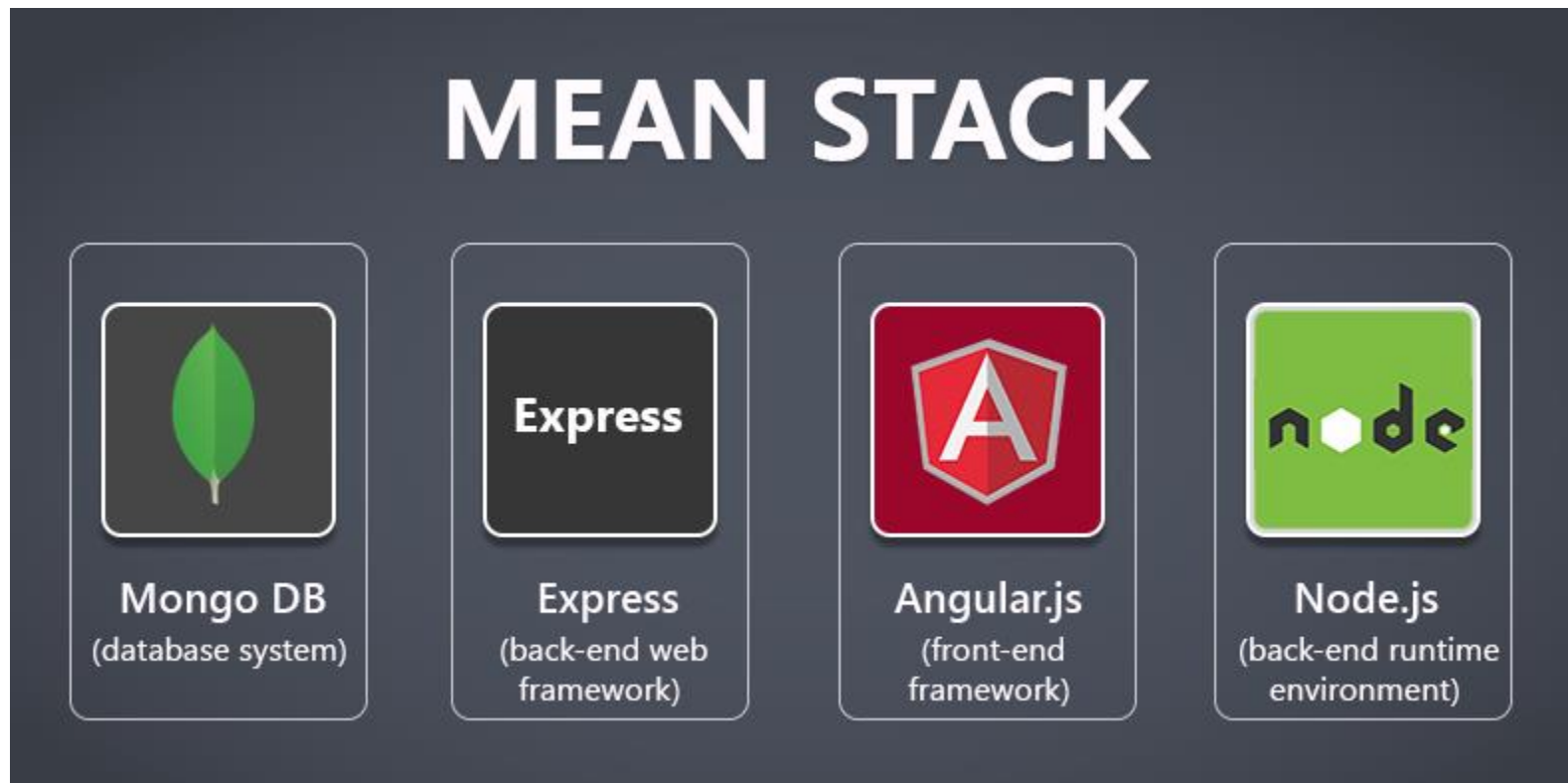
MONGODB:

POPULARITY

Rank			DBMS	Database Model	Score		
May 2021	Apr 2021	May 2020			May 2021	Apr 2021	May 2020
1.	1.	1.	Oracle	Relational, Multi-model	1269.94	-4.98	-75.50
2.	2.	2.	MySQL	Relational, Multi-model	1236.38	+15.69	-46.26
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	992.66	-15.30	-85.64
4.	4.	4.	PostgreSQL	Relational, Multi-model	559.25	+5.73	+44.45
5.	5.	5.	MongoDB	Document, Multi-model	481.01	+11.04	+42.02
6.	6.	6.	IBM Db2	Relational, Multi-model	166.66	+8.88	+4.02
7.	7.	8.	Redis	Key-value, Multi-model	162.17	+6.28	+18.69
8.	8.	7.	Elasticsearch	Search engine, Multi-model	155.35	+3.18	+6.23
9.	9.	9.	SQLite	Relational	126.69	+1.64	+3.66
10.	10.	10.	Microsoft Access	Relational	115.40	-1.33	-4.50
11.	11.	11.	Cassandra	Wide column	110.93	-3.92	-8.22
12.	12.	12.	MariaDB	Relational, Multi-model	96.69	+0.32	+6.61
13.	13.	13.	Splunk	Search engine	92.11	+3.62	+4.36
14.	14.	14.	Hive	Relational	76.19	-2.31	-5.35
15.	15.	23.	Microsoft Azure SQL Database	Relational, Multi-model	70.46	-1.39	+27.70
16.	16.	16.	Amazon DynamoDB	Multi-model	70.07	-0.66	+5.35
17.	17.	15.	Teradata	Relational, Multi-model	69.98	-0.57	-3.91
18.	18.	20.	SAP HANA	Relational, Multi-model	52.75	-0.69	+2.22
19.	20.	21.	Neo4j	Graph	52.23	+1.19	+2.47
20.	21.	18.	Solr	Search engine, Multi-model	51.19	+0.59	-1.39
21.	19.	17.	SAP Adaptive Server	Relational, Multi-model	49.96	-1.70	-4.03
22.	22.	19.	FileMaker	Relational	46.73	+0.32	-4.23
23.	23.	22.	HBase	Wide column	43.24	-0.92	-6.48
24.	24.	26.	Google BigQuery	Relational	37.63	+2.05	+10.04
25.	25.	24.	Microsoft Azure Cosmos DB	Multi-model	34.71	+1.19	+4.03

MEAN Stack

- MongoDB, Express.js, Angular(JS), Node.js





Questions?