NodeJS
środowisko i technologia ServerSide

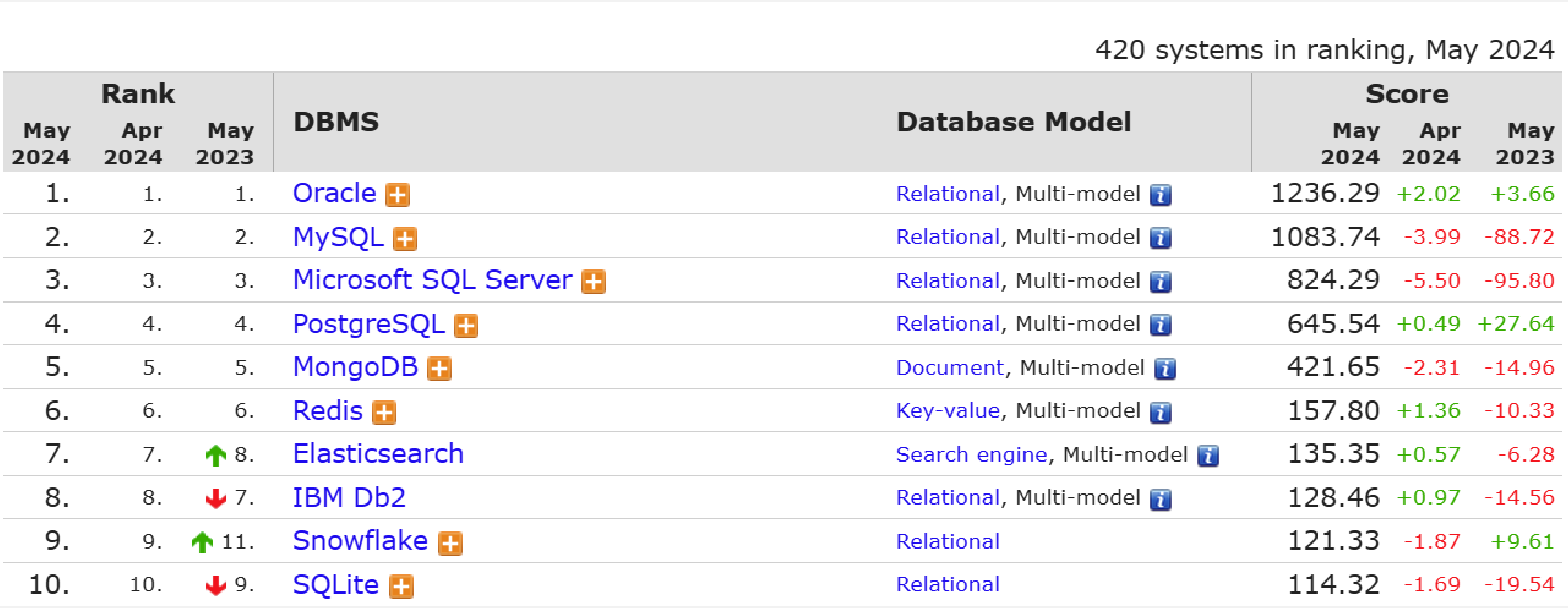
PAWEŁ ŁUKASZUK

Rok akademicki: 2023/2024, Semestr: 2, Wykład: 4

71
HEAT SET TO
63

Tempesta
7
6

MongoDB



420 systems in ranking, May 2024

Rank			DBMS	Database Model	Score		
May 2024	Apr 2024	May 2023			May 2024	Apr 2024	May 2023
1.	1.	1.	Oracle	Relational, Multi-model	1236.29	+2.02	+3.66
2.	2.	2.	MySQL	Relational, Multi-model	1083.74	-3.99	-88.72
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	824.29	-5.50	-95.80
4.	4.	4.	PostgreSQL	Relational, Multi-model	645.54	+0.49	+27.64
5.	5.	5.	MongoDB	Document, Multi-model	421.65	-2.31	-14.96
6.	6.	6.	Redis	Key-value, Multi-model	157.80	+1.36	-10.33
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	135.35	+0.57	-6.28
8.	8.	↓ 7.	IBM Db2	Relational, Multi-model	128.46	+0.97	-14.56
9.	9.	↑ 11.	Snowflake	Relational	121.33	-1.87	+9.61
10.	10.	↓ 9.	SQLite	Relational	114.32	-1.69	-19.54

https://db-engines.com/en/ranking

Password encoding

Connection string used by MongoDB is ultimately just an example of URI

mongodb+srv://login:pass@cluster.mongodb.net/?retryWrites=true

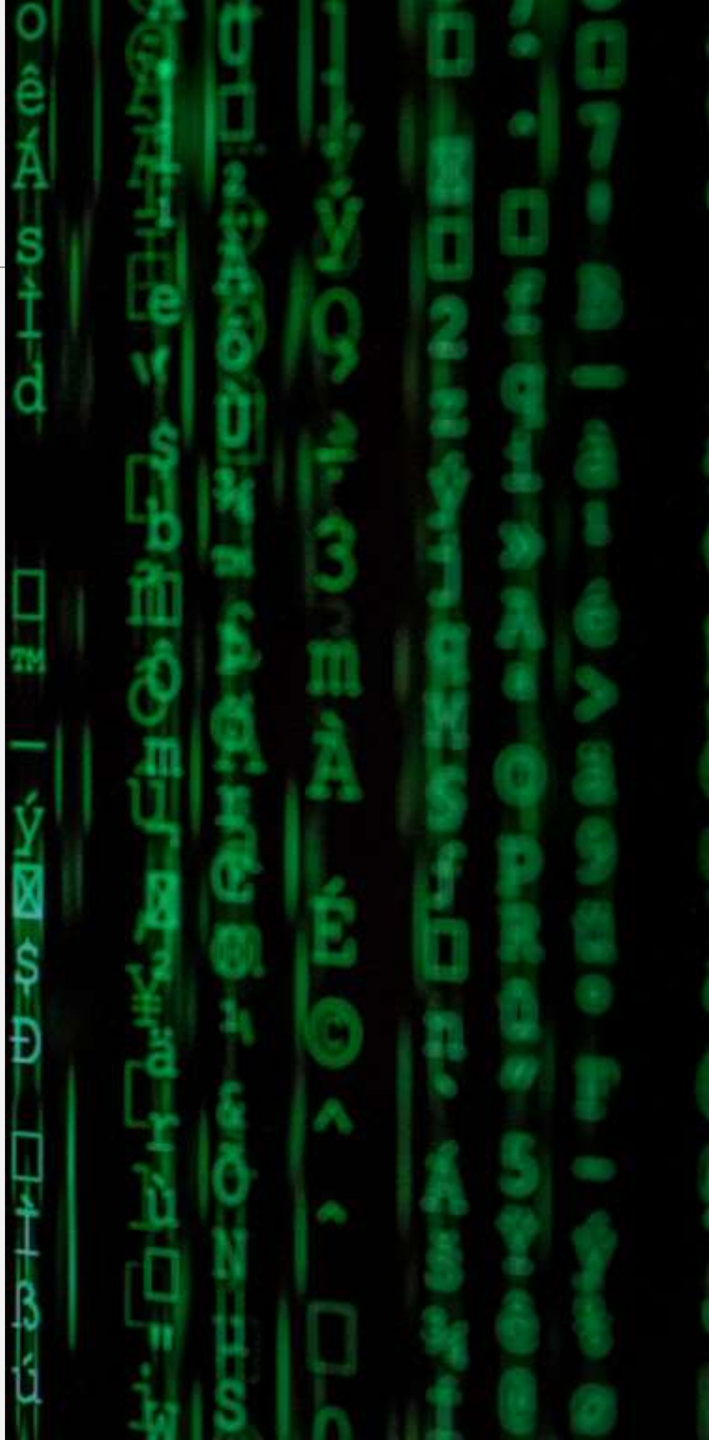
This is why some special characters in login and password needs to be encoded – replaced by special sequence.

For instance:

- / → %2F
- : → %3A
- @ → %40

This can be done using any URL encoding tool

or with standard JavaScript function: encodeURIComponent



CRUD

- Create
- Read
- Update
- Delete

Create

- db.collection.insertOne()
- db.collection.insertMany()

All write operations in MongoDB are atomic on the level of a single document.

If collection does not exist, insert operations will create new collection.

If an inserted document omits the _id field the MongoDB driver automatically generates an ObjectId for the _id field

Create

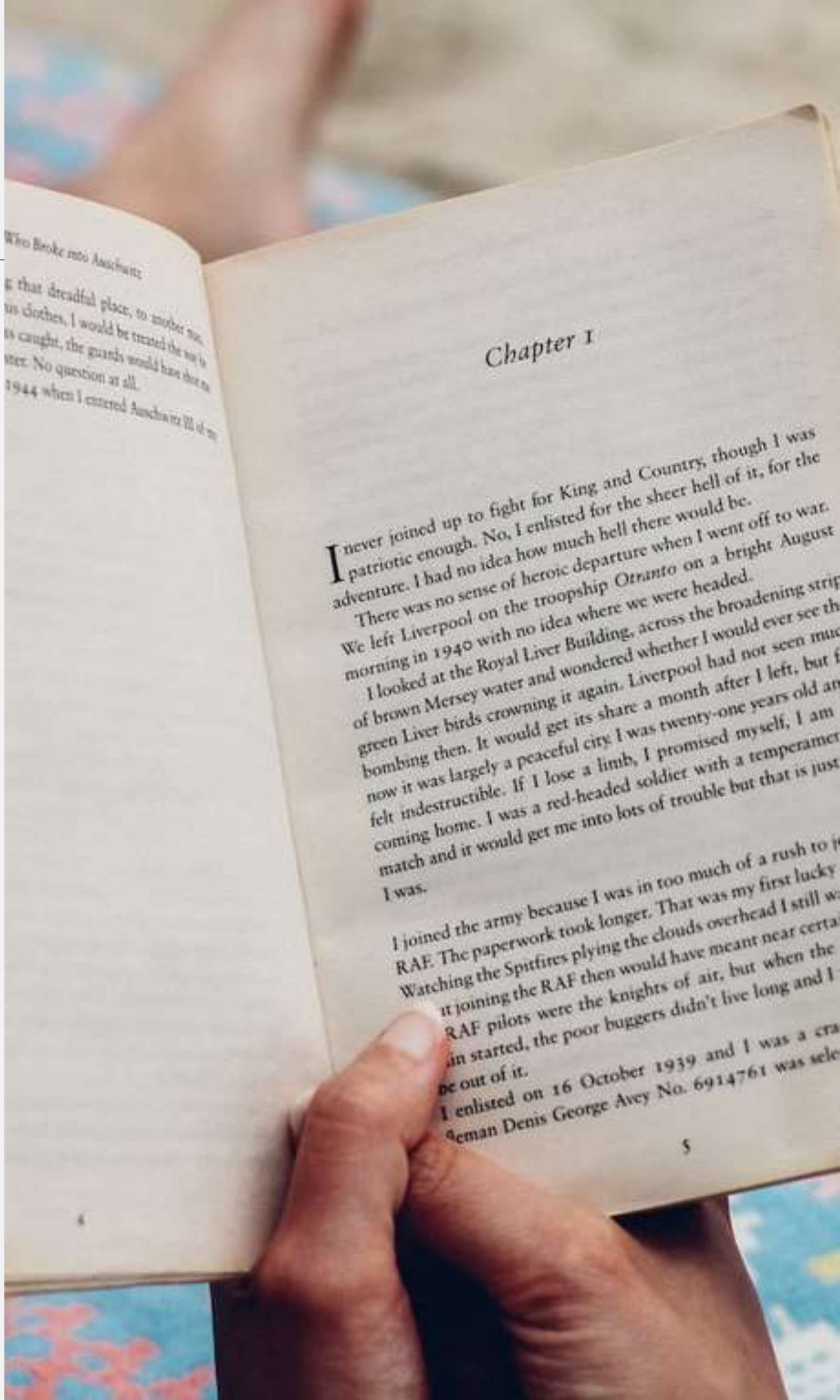
```
db.collection.insertOne({
    firstName: 'Jan',
    lastName: 'Kowalski'
});

db.collection.insertMany([{
    firstName: 'Jan1',
    lastName: 'Kowalski1'
},{
    firstName: 'Jan2',
    lastName: 'Kowalski2'
}]);
```

Read

- db.collection.findOne()
- db.collection.find()

If multiple documents match the criteria, findOne function returns the first document according to the order the documents are stored on disk.



Read

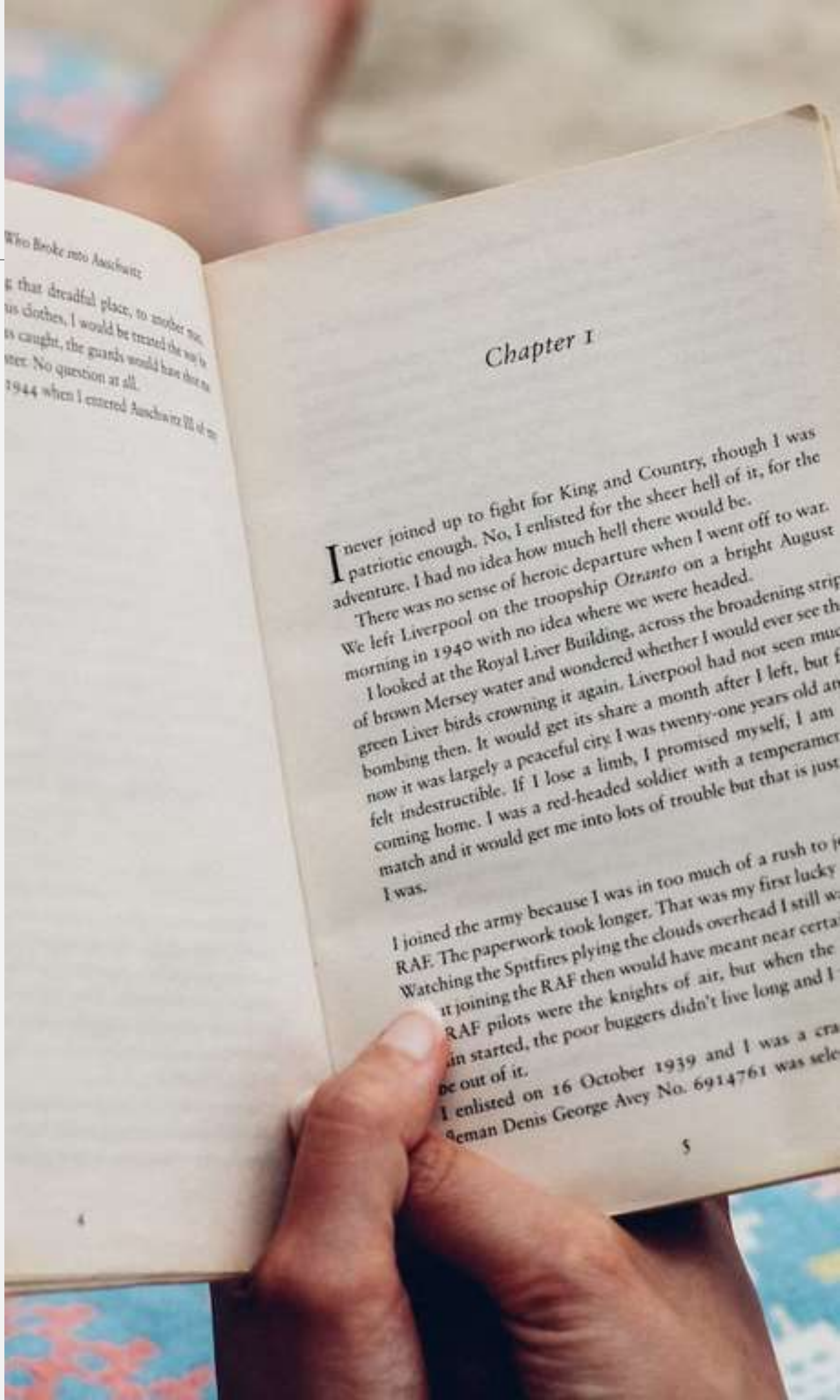
```
const user = await db.collection.findOne();

console.table(user);


const users = await db.collection.find()

                                 .toArray();

console.table(users);
```



Read

Find functions can be extended with:

- complex search conditions: Query
- instruction to retrieve only part od documents: Projection
- cursor functions to get data about multple results (sort, skip, limit)

```
db.collection.findOne({ query });

db.collection.find({ query })
              .project({ projection })
              .cursorFunction();
```

Read – query

Find method supports many different query operators to filter data

- comparison
- logical
- element
- evaluation
- geospatial
- array
- bitwise

```
{ field : { $operator : value } }
```

Read – query operators

Comparison query operators

• $eq	=	// equal
• $ne	!=	// not equal
• $in	element in given array	// in
• $nin	element is not in given array	// not in
• $lt	<	// lesser than
• $lte	<=	// lesser than or equal
• $gt	>	// greater than
• $gte	>=	//greater than or equal

Read – query operators

```
db.collection.findOne({ lastName : { $eq : "Kowalski" }});
db.collection.find({ age : { $gt : 15 }});
```

Instead of equality operator $eq we can use short syntax:

```
{ field : value }  // is equal to  { field : { $eq : value } }

db.collection.findOne({ lastName : "Kowalski" });
// is equal to
db.collection.findOne({ lastName : { $eq : "Kowalski" } });
```

Read – query operators

To check multiple conditions we can combine multiple checks with logical operators

- $and
- $or

```
{ logical_operator : [{firstCondition},{secondCondition}] }

{ $or : [{ firstName: "Adam" }, { lastName: "Kowalski" }] }
{ $and : [{ firstName: "Adam" }, { lastName: "Kowalski" }] }
```

Read – query operators simplifications

And operation can be simplified to syntax:

```
{ $and : [{ firstName: "Adam" }, { lastName: "Kowalski" }] }
// is equal to
{ firstName: "Adam", lastName: "Kowalski" }
```

Short syntax, only when using same field in all expressions

```
{ $and: [ { age: {$lt : 40 }} , { age: {$gt : 30 }} ] }
// is equal to
{ age : { $lt:40, $gt:30 } }
```

Read – query with a bit of complexity

Operators can be combined in more complex manner:

```
{ $and: [
        { firstName: { $eq: "Adam" } },
        { $or: [ { age: { $gt: 20 }} ,
                 { height: { $lt: 200 } } ] },
        ]
}

// could be explained as:
// ( firstName == "Adam" and (age > 20 or height < 200) )
```

Read – query and nested objects

To access nested objects is necessary to wrap field names with ”” signs

```
// given document in database
 {
     "_id": { "$oid": "64287a5f57d7d23554a7f499" },
     "firstName": "Jan2",
     "lastName": "Kowalski2",
     "isActive": true,
     "address": { "postCode": "00-000" }
 }

 // can be found using query
 find({ "address.postCode" : "00-000"})
```

Read – query with null

```
find({ address : null})
```

This query will returns documents where address is null or address does not exists.

Operator $exists can be used to query documents where a field exists or not, regardless of its value.

```
{ address : { $exists : false } } // address does not exists
```

```
{ address : { $exists : true } } // address exists (can be null or has value)
```

Read – query by type

Operator $type can be used to query documents where the value of a field is of a specified BSON type.

There is also the “number” alias which can match against all numeric types (double, 32-bit integer, 64-bit integer, decimal).

Type	Number	Alias	Notes
Double	1	"double"	
String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary data	5	"binData"	
Undefined	6	"undefined"	Deprecated.
ObjectId	7	"objectId"	
Boolean	8	"bool"	
Date	9	"date"	
Null	10	"null"	
Regular Expression	11	"regex"	
DBPointer	12	"dbPointer"	Deprecated.
JavaScript	13	"javascript"	
Symbol	14	"symbol"	Deprecated.
JavaScript code with scope	15	"javascriptWithScope"	Deprecated in MongoDB 4.4.
32-bit integer	16	"int"	
Timestamp	17	"timestamp"	
64-bit integer	18	"long"	
Decimal128	19	"decimal"	
Min key	-1	"minKey"	
Max key	127	"maxKey"	

Read – query by type

```
db.collection.find({ firstName : { $type: "string" }});


// is equal to
db.collection.find({ firstName : { $type: 2 }});
```

Read – projection

Projection controls which fields appear in the documents returned by read operations. Projections can help you limit unnecessary network bandwidth usage.

Projections work in two ways:

- include fields with a value of 1. This has the side-effect of implicitly excluding all unspecified fields.
- exclude fields with a value of 0. This has the side-effect of implicitly including all unspecified fields.

These two methods of projection are mutually exclusive:

if you explicitly include fields, you cannot explicitly exclude fields, and vice versa.

Read – projection

Projection specifies the fields to return in the documents that match the query filter

- 1 or true : include the field
- 0 or false : exclude the field

By default _id field is always returned

```
// return all fields except firstName
db.collection.find().project({ firstName : 0 });

// return firstName and _id
db.collection.find().project({ firstName : 1 });
```

Read – projection

```
// return only firstName
db.collection.find().project({ firstName : 1, _id : 0 });

// return only firstName, lastName, and _id
db.collection.find().project({ firstName : 1, lastName : 1 });

// return only firstName and lastName
db.collection.find()
             .project({firstName : true, lastName : true, _id : false});
```

Read – sort

Sort changes order in which read operations return documents.

To sort returned documents by a field in ascending (lowest first) order, use a value of 1.

To sort in descending (greatest first) order instead, use -1.

If you do not specify a sort, MongoDB does not guarantee the order of query results.

Read – sort

```
// sort by firstName ascending
db.collection.find().sort({ firstName: 1 });

// sort by firstName descending
db.collection.find().sort({ firstName: -1 });

// sort by lastName ascending and then by firstName ascending
db.collection.find().sort({ lastName: 1, firstName: 1 });
// if two users have same lastName first one will be this one
// which firstName is first in alphabetic order
```

Read – skip

Skip omits documents from the beginning of the list of returned documents for a read operation.

You can combine skip with sort to omit the top (for descending order) or bottom (for ascending order) results for a given query.

```
db.collection.find().skip(1);
```

Read – limit

Cap the number of documents that can be returned from a read operation.

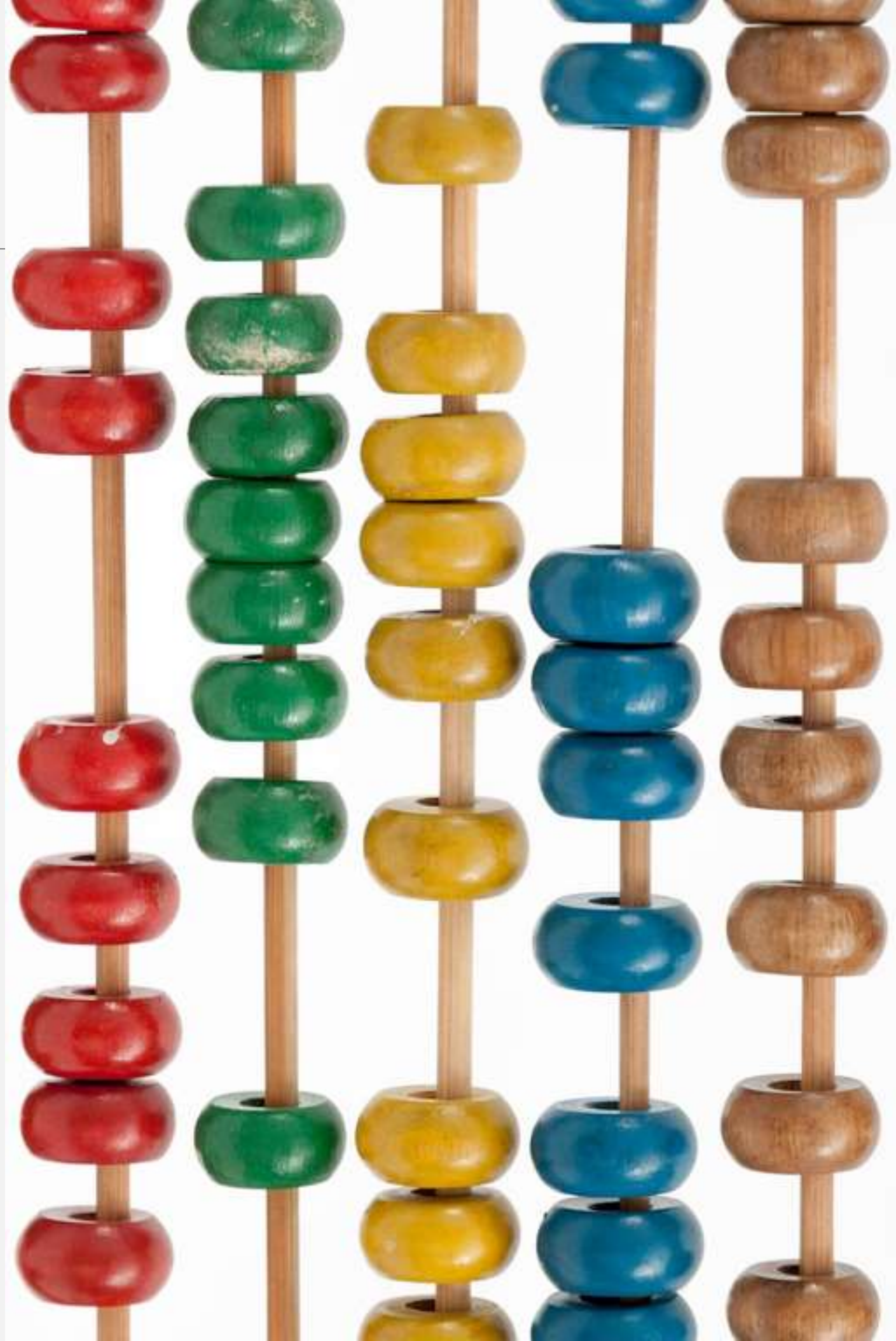
Limit functions as a cap on the maximum number of documents that the operation can return, but the operation can return a smaller number of documents if there are not enough documents present to reach the limit.

```
db.collection.find().limit(2);
```

Read – count

The Node.js driver provides two methods for counting documents in a collection:

- countDocuments() – returns accurate number of documents based on given query or numer of all documents when query is empty
- estimatedDocumentCount() – returns estimated number of documents based on collection metadata, does not accept query



Read – count

```
// estimated numer of documents in collection
db.collection.estimatedDocumentCount();


// number of documents in collection
db.collection.countDocuments();


// number of documents in collection that fulfills given condition
db.collection.countDocuments({ lastName : "Kowalski" });
```

Update

- db.collection.updateOne()
- db.collection.updateMany()

Update operation is atomic on the level of a single document

- _id field cannot be replaced with different value
- $set creates field if not already existing

Update

```
// set firstName as ”Marek” in all documents

db.collection.updateMany({}, { $set: { firstName: 'Marek' } });


// set firstName as ”Marek” in first document

db.collection.updateOne({}, { $set: { firstName: 'Marek' } });


// set firstName as ”Marek” and isActive as true in first document where
lastName is Kowalski

db.collection.updateOne({ lastName: 'Kowalski’ },

                        { $set: { firstName: 'Marek', isActive: true } }

);
```

Update – upsert

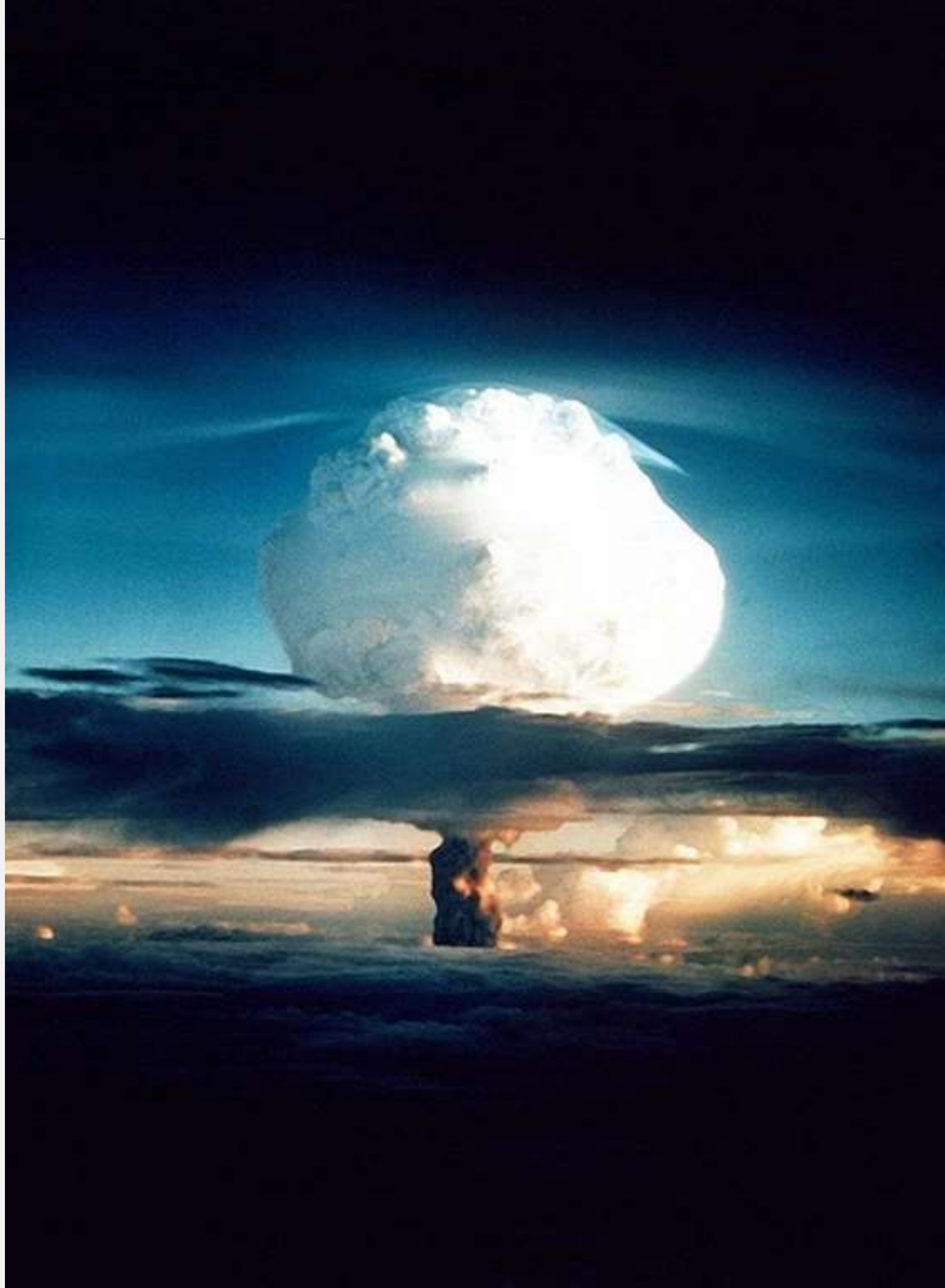
Upsert - update on match of filter or insert no match of filter. By default is set to false.

```
// if no document match query then those functions will create one document
// { lastName: ”Kowalski”, firstName: ”Marek”, isActive: true }
db.collection.updateOne({ lastName: 'Kowalski’ },
                          { $set: { firstName: 'Marek', isActive: true } },
                          { upsert : true });

db.collection.updateMany({ lastName: 'Kowalski’ },
                          { $set: { firstName: 'Marek', isActive: true } },
                          { upsert : true });
```

Delete

- db.collection.deleteOne()
- db.collection.deleteMany()



Delete

```
// delete first document with firstName equal ”Marek”
db.collection.deleteOne( { firstName: 'Marek' } );

// delete all documents
db.collection.deleteMany();
```

Collection operations

Create collection:

```
db.createCollection("Users");
```

Drop collection:

```
db.collection.drop();
```

Index support

Indexes are data structures that support the efficient execution of queries in MongoDB. They contain copies of parts of the data in documents to make queries more efficient.

```
db.collection.createIndex(
    { firstName: 1, lastName: 1 }
);
```