Focus on queries

What is MongoDb

- MongoDB is an open-source NoSQL database
  - Document oriented
  - High performance
  - High availability
  - Horizontal scalability (sharding)
- Definitions
  - Collection: a group of documents
  - Document: a set of key-value pairs

This introduction is deeply inspired from https://www.tutorialspoint.com/mongodb/mongodb_tutorial.pdf

MongoDB - concepts

- Think of **documents** as database records
  - Documents are just JSON objects that MongoDB stores in binary (BSON format)
- Think of **collections** as database tables

RDBMS (mysql, postgresql)	MongoDB
Database	Database
Table	Collection
Record/row	Document/object
Column	Field
Queries return a record	Queries return a **cursor**

mongoDB.

MongoDB - concepts

mongoDB.®

MongoDB as repository

- Queries return ”cursors” instead of a collections
  - A cursor allows you to iterate through the result set
  - A big reason for this is performance
  - Much more efficient to load results into memory
    - Especially if results are big as in big data

- The find() function returns a **cursor** object

```
var c = db.ActiveBookings.find( {city: "Torino"} ) // c is the cursor
var i = 0
while (c.hasNext() && i<10)
{
  var o = c.next() // o is the object
  print(o.init_time + " " + o.city)
  i++
}
```

mongoDB.®

A sample document

```
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100,
comments: [
    {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2011,1,20,2,15)
    },
    {
        user:'user2',
        message: 'My second comments',
        dateCreated: new Date(2011,1,25,7,45),
    }
    ]
}
```

A sample document

```
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100,
comments: [
    {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2011,1,20,2,15)
    },
    {
        user:'user2',
        message: 'My second comments',
        dateCreated: new Date(2011,1,25,7,45),
    }
    ]
}
```

Document

mongoDB.

A sample document

```
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100,
comments: [
    {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2011,1,20,2,15)
    },
    {
        user:'user2',
        message: 'My second comments',
        dateCreated: new Date(2011,1,25,7,45),
    }
    ]
}
```

Key : Value

_id is the unique identifier for each object in the DB
Added by Mongo during insert, and automatically indexed.
It is of type ObjectId

A sample document

```
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100,
comments: [
    {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2011,1,20,2,15)
    },
    {
        user:'user2',
        message: 'My second comments',
        dateCreated: new Date(2011,1,25,7,45),
    }
    ]
}
```

Key : Value

Another **Key: Value**
Key is **title**, and
value is a **string**

A sample document

```
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100,
comments: [
    {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2011,1,20,2,15)
    },
    {
        user:'user2',
        message: 'My second comments',
        dateCreated: new Date(2011,1,25,7,45),
    }
    ]
}
```

Key : Value

tags is a key, whose value
is an **ARRAY**
Arrays are list of values,
enclosed between **[ ]**

Note:These are compact representation of
embedded objects,
with integer values for the keys,
starting with 0 and continuing sequentially
For example, the array ['red', 'blue']
is equivalent to the document {'0': 'red', '1': 'blue'}

A sample document

```
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100,
comments: [
    {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2011,1,20,2,15)
    },
    {
        user:'user2',
        message: 'My second comments',
        dateCreated: new Date(2011,1,25,7,45),
    }
    ]
}
```

Key : Value

Comments is another key
Whose values is an **array**
Whose elements are **embedded objects**

A sample document

```
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
description: 'MongoDB is no sql database',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 100,
comments: [
    {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2011,1,20,2,15)
    },
    {
        user:'user2',
        message: 'My second comments',
        dateCreated: new Date(2011,1,25,7,45),
    }
    ]
}
```

Key : Value

Each object has three key-value pairs
user and **message** contains **strings**
dateCreated is a type of **Date**

mongoDB.

Datatypes – part I

- **String**: This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid (Unicode Transformation Format, 8 bit).
- **Integer**: This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean**: This type is used to store a boolean (true/ false) value.
- **Double**: This type is used to store floating point values.
- **Min/Max Keys**: This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays**: This type is used to store arrays or list or multiple values into one key.
- **Timestamp**: ctimestamp. This can be handy for recording when a document has been modified or added.

Datatypes – part II

- **Object**: This datatype is used for embedded documents.
- **Null**: This type is used to store a Null value.
- **Symbol**: This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date**: This datatype is used to store the current date or time in **UNIX time format**. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID**: This datatype is used to store the document's ID.
- **Binary data**: This datatype is used to store binary data.
- **Code**: This datatype is used to store JavaScript code into the document.
- **Regular expression**: This datatype is used to store regular expression.

Advantages of MongoDB vs RDBMS

- Schema less: MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another
- Structure of a single object is clear.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Ease of scale-out: MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why using MongoDB

- Document Oriented Storage: Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-sharding
- Rich queries
- Fast in-place updates

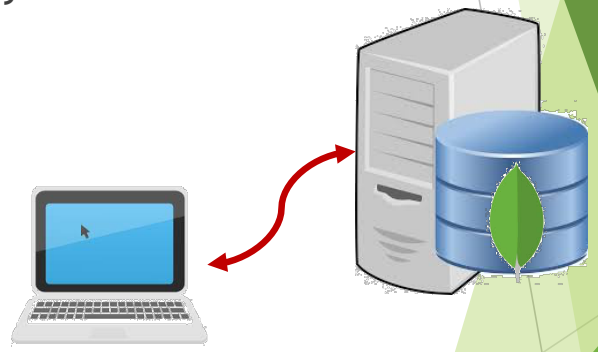
Data modelling

- Data in MongoDB has a flexible **schema.documents** in the same collection
  - They do not need to have the same set of fields or structure
  - Common fields in a collection's documents may hold different types of data
  - Indexes can be added at any time to speed up query

Our CarSharing MongoDB

Access to a MongoDB

- In the typical scenario, the MongoDB **server** (mongod) runs in the backend
- You connect to it using **clients** running on different system
  - There are different interfaces to access to the DB
    - The Mongo Shell
      - An interactive JavaScript interface to MongoDB
    - Robomongo - www.robomongo.org
      - Same as the the Mongo Shell, but with a nice GUI
    - mongodb Compass - docs.mongodb.com/compass
      - A tool that helps you visually analyse and understand your MongoDB data
    - YMMV - https://www.google.it/?q=mongodb+client+gui

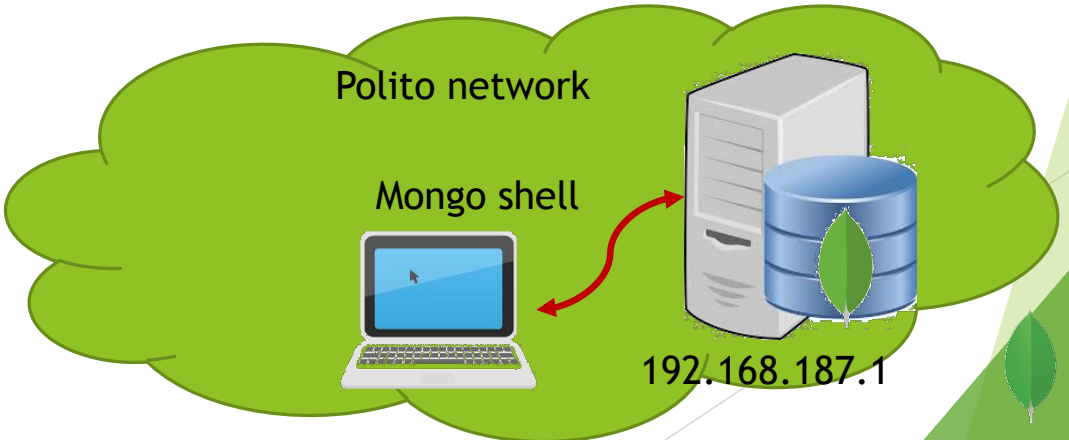


mongoDB.

Access to the Carsharing DB

- Our server runs on a VM @ 192.168.187.1
  - Private address -- can only be accessed from Polito network
- Requires authentication to connect to the **`Carsharing`** database
  - User: **`ictts`** - Pwd: **`Ictts16!`**

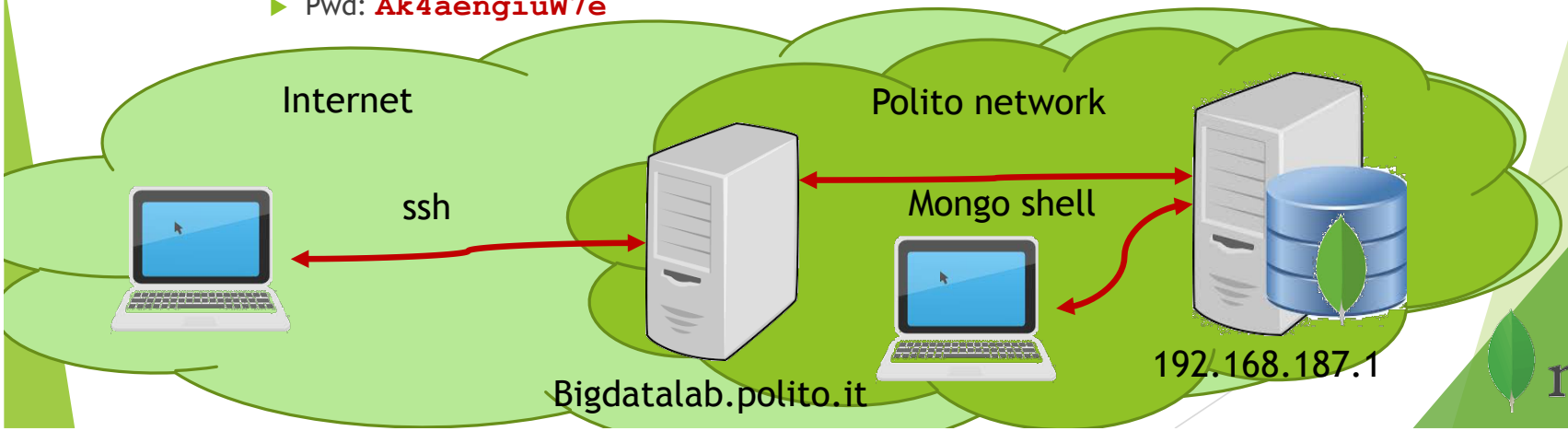
```
mongo 192.168.187.1/Carsharing -u 'ictts' -p 'Ictts16!'
```



mongoDB.

Access to the Carsharing DB

- The server runs on a VM @ 192.168.187.1
  - Private address -- can only be accessed from Polito network
    - It's likely to not work over WiFi due to firewall restriction
  - You can access **`bigdata.polito.it`** public server via ssh
    - Username: **`s100001`**
    - Pwd: **`Ak4aengiuW7e`**

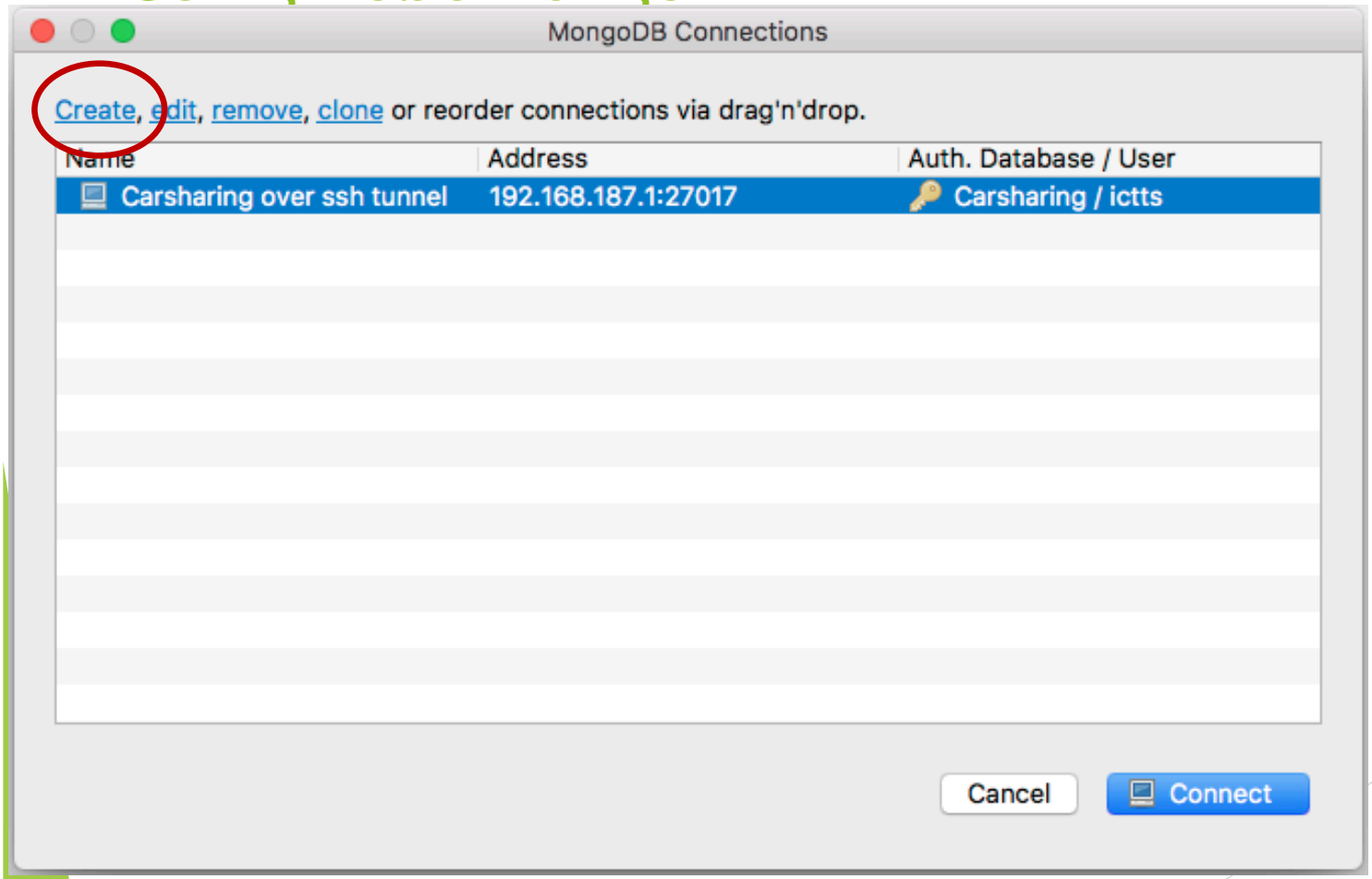


mongoDB.

Access to the Carsharing DB

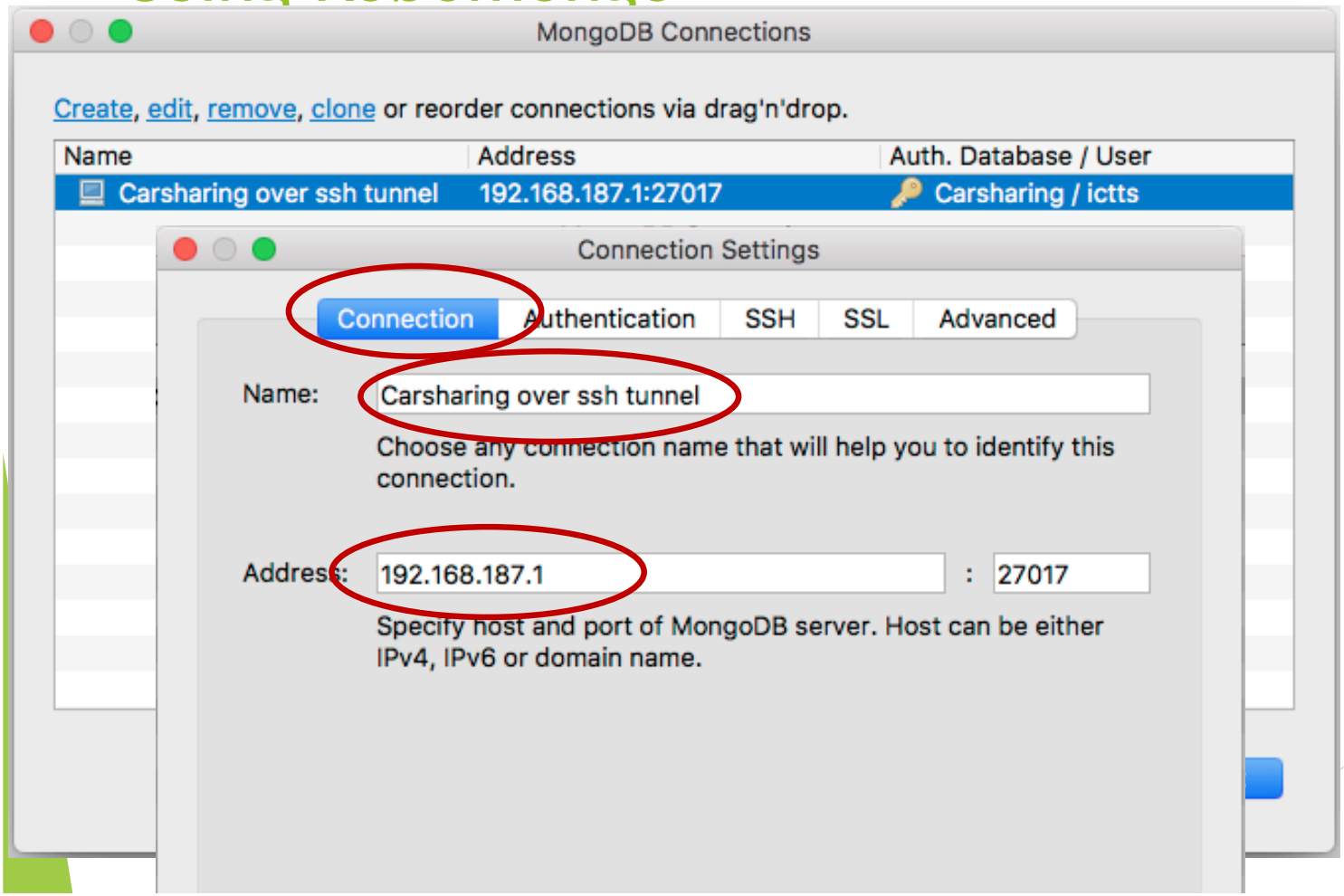
```
MacMGM-456:~ mellia$ ssh s100001@bigdatalab.polito.it
100001@bigdatalab.polito.it's password: [ Ak4aengiuW7e ]
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.4.0-38-generic x86_64)
[…]
Last login: Thu Dec 29 17:52:04 2016 from 192.168.185.2
s100001@bigdatalab:~$ mongo 192.168.187.1/Carsharing -u ictts -p "Ictts16!"
MongoDB shell version: 2.4.9
connecting to: 192.168.187.1/Carsharing
> db
Carsharing
> db.stats()
{
    "db" : "Carsharing",
    "collections" : 6,
    "objects" : 2665670,
     […]
}
>
```

Using Robomongo



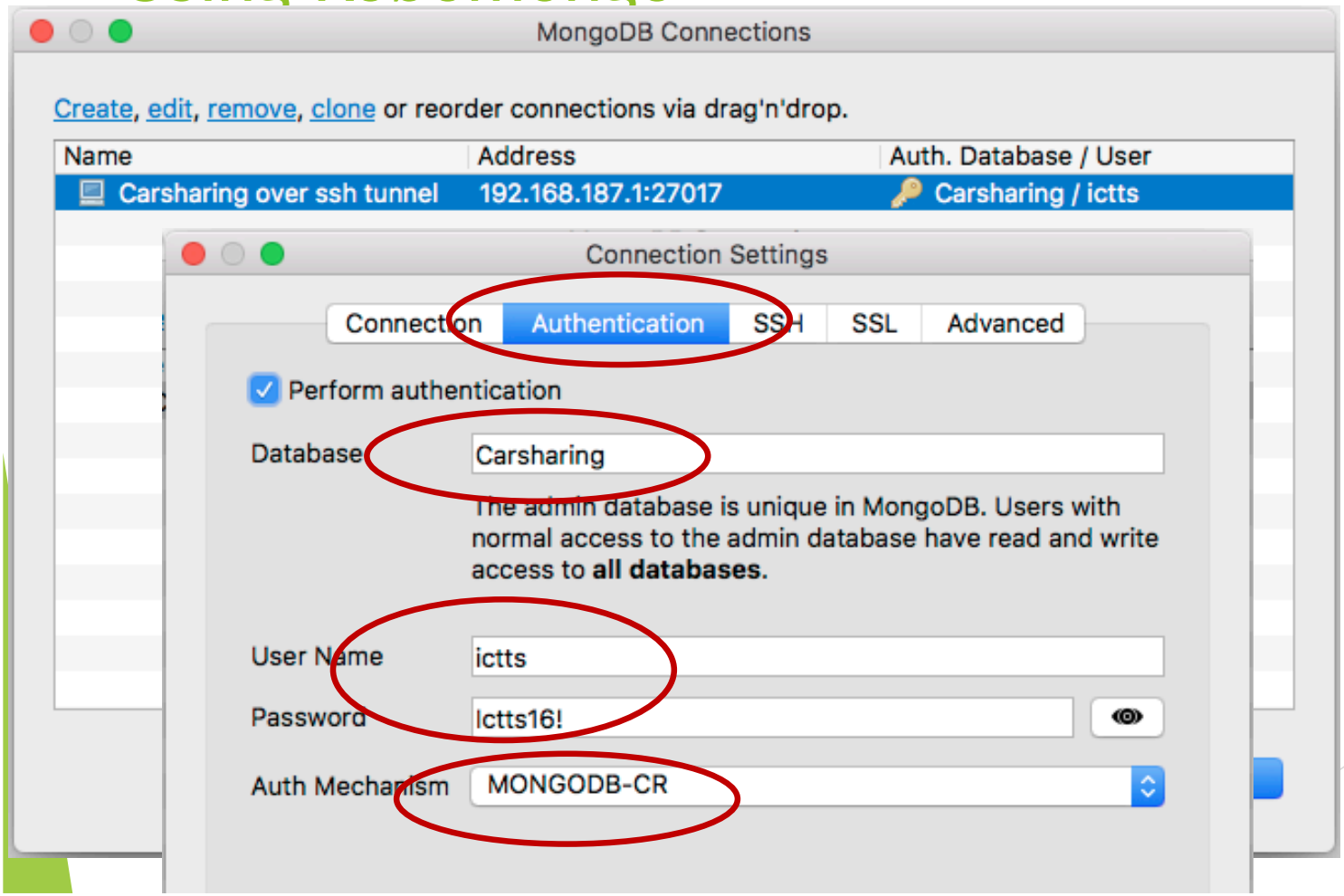
mongoDB.

Using Robomongo



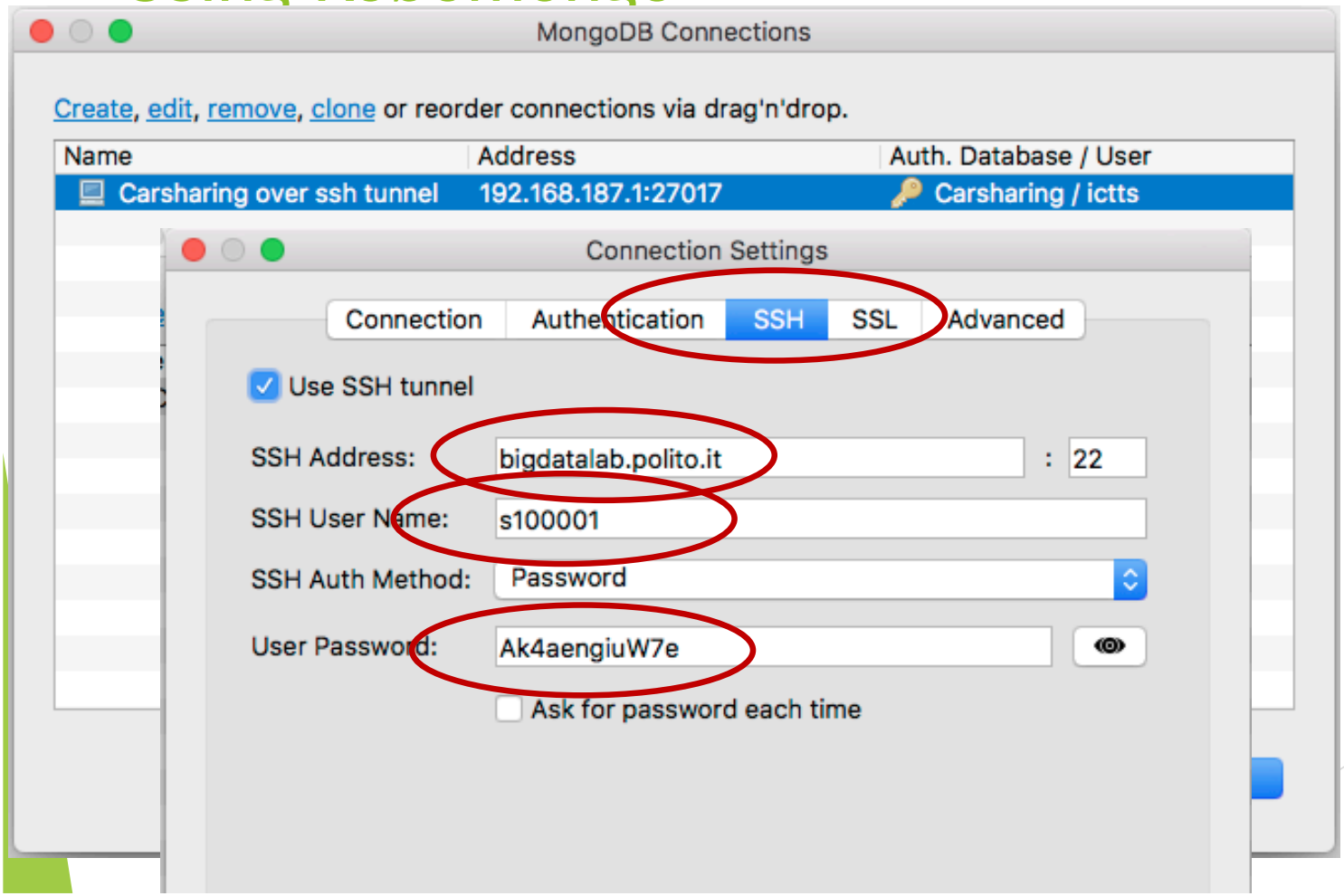
mongoDB.

Using Robomongo



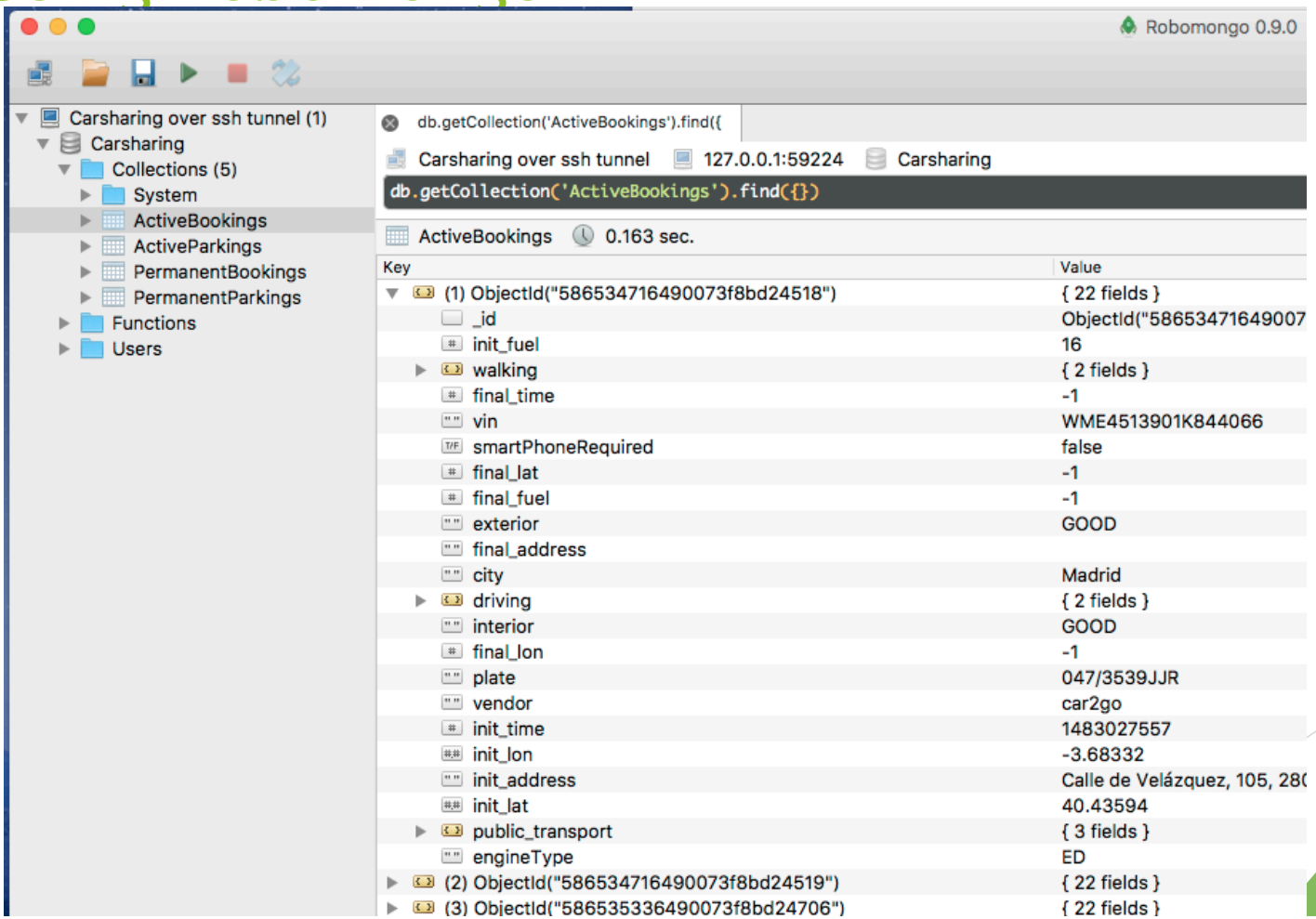
mongoDB.

Using Robomongo



mongoDB.

Using Robomongo



mongoDB.

Access to the Carsharing DB

- The Carsharing DB contains 4 collections

```
> db.getCollectionNames()
[
    "ActiveBookings",
    "ActiveParkings",
    "PermanentBookings",
    "PermanentParkings",
    "system.indexes"
]
```

Show the name of the collections in this dababase

Contains cars that are currently booked

Contains cars that are currently parked

Contains all booking seen so far

Contains all parking seen so far

[Additional collection to handle indexes]

The Mongo Shell

Access to the Carsharing DB

- The **`mongo`** shell is an interactive **JavaScript** interface to MongoDB
  - You can use the **`mongo`** shell to query and update data as well as perform administrative operations
  - Offers
    - TAB completion
    - Online help
      ```
      db.help()
      db.collection.help()
      db.collection.find().help()
      ```
  - You get a **READ-ONLY** access to the **`Carsharing`** DB only
    - Can **`find()`**, **`aggregate()`**, …
    - Cannot **`insert()`**, **`drop()`**, **`update()`**, …

Executing a javascript file

- Mongo uses javascript as language
- You can interact with the Mongo shell directly

```
mongo 192.168.187.1/Carsharing –u”ictts” –p “Ictts16!”
> db.activebookings.find().count()
3249
```

- Or you can execute javascripts

```
echo "var tot = db.ActiveBookings.find().count()" > count.js
echo "print (tot)" >> count.js
mongo 192.168.187.1/Carsharing -u 'ictts' -p 'Ictts16!' count.js
MongoDB shell version v3.4.0
connecting to: mongodb://192.168.187.1/Carsharing
MongoDB server version: 2.6.12
3204
```

Edit a file

Execute the script

Get the result

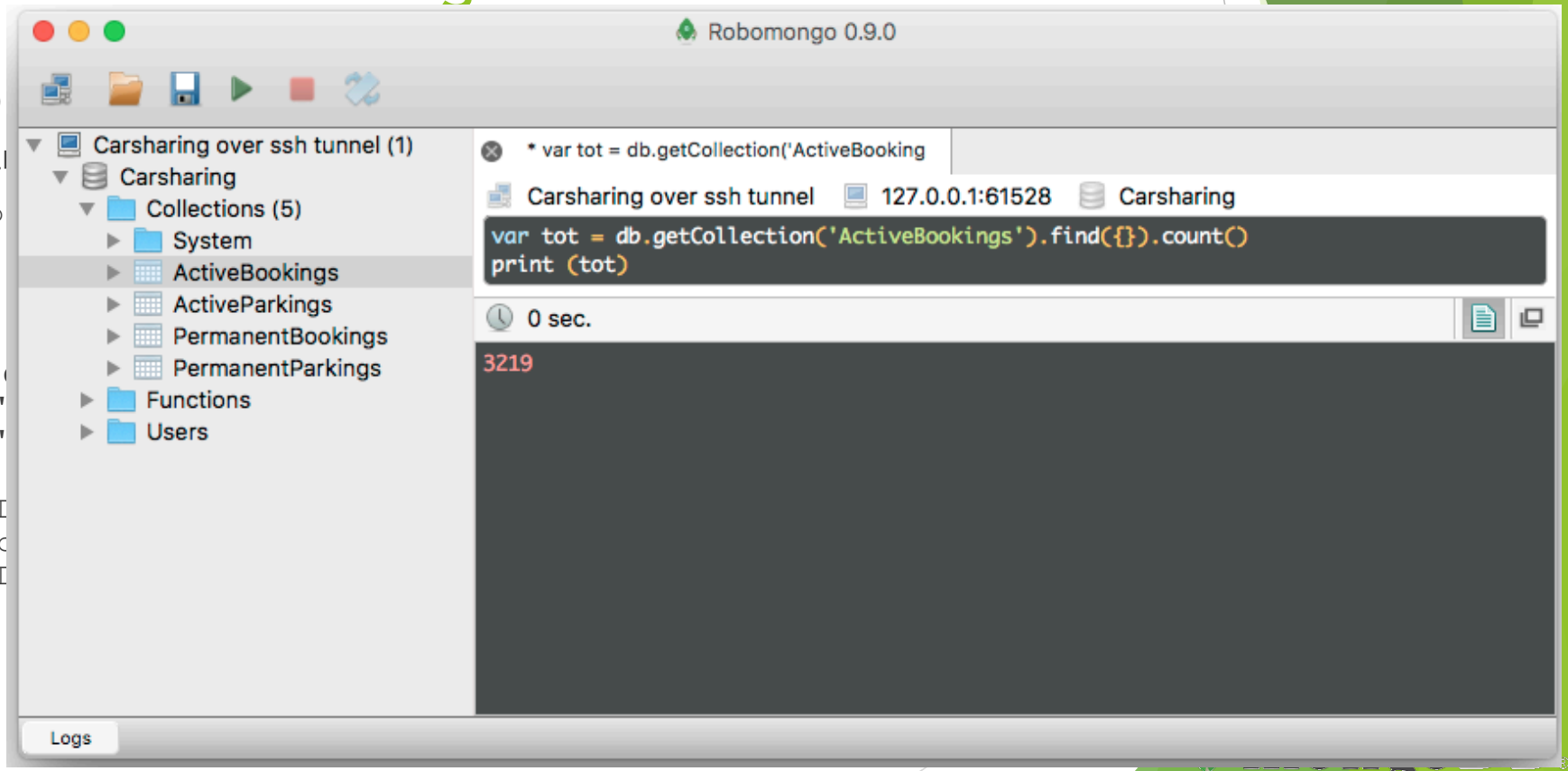
Better using robomongo… then save the script for later usage…

- Mongo
- You ca

```
mongo
> db.
3249
```

- Or you

```
echo '
echo '
mongo
MongoD
connec
MongoD
3204
```



Mongo Query

CRUD operations

- CRUD operations on documents
  - *Create*
    - *db.collection.insert()*
  - *Read*
    - ***db.collection.find()***
  - *Update*
    - *db.collection.update()*
  - *Delete*
    - *db.collection.remove()*

The find() method

Read operations – the find() method

- Syntax

  `db.COLLECTION_NAME.find(<query>)`

  - `COLLECTION_NAME` is the name of the collection over which to apply the `find()` method
- Example

  `db.ActiveBookings.find()`

  - Returns all object in the ActiveBookings collection in the current db
- Useful methods
  - `.pretty()` => print in a formatted way
  - `.findone()` => returns only one document
  - `.limit(<n>)` => returns the first n entries
  - `.skip(<n>)` => returns the documents after the first n entries
  - `.count()` => returns the number of matches
  - `.forEach(<function>)` => Iterates the cursor to apply a JavaScript `function` to each document from the cursor

MongoDB - concepts

- Recalll: Queries return **cursors** instead of a collections
- The find() function returns a **cursor** object

```
var c = db.ActiveBookings.find( {city: "Torino"} ) // c is the cursor
var i = 0
while (c.hasNext() && i<10) {
  var o = c.next() // o is the object
  print(o.init_time + " " + o.city)
  i++
}
```

- This can be written in a more compact way

```
db.ActiveBookings.find({city: "Torino"}).limit(10).forEach(function(o){
    print( o.init_time + ” “ + o.city)
    })
```

- Question:
  - Which one is faster???
  - In which order are documents returned?

Example of document

```
db.ActiveBookings.findOne()
{
    "_id" : ObjectId("5863c3246490073f8bcfa100"),
    "init_fuel" : 24,
    "walking" :
     {
        "duration" : -1,
        "distance" : -1
      },
    "final_time" : -1,
    "vin" : "WMEEJ3BA4EK748306",
    "smartPhoneRequired" : false,
    "final_lat" : -1,
    "final_fuel" : -1,
    ”city" : ”Torino",
    …
    }
```

Note: is equivalent to

```
db.ActiveBookings.find().limit(1).skip(0)
```

mongoDB.

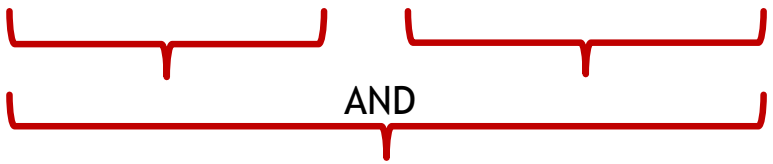
Conditions

- You can specify **query filters** or criteria that identify the documents to return
  `db.COLLECTION_NAME.find(<query>)`
- A `<query>` **filter document** can specify **equality** condition with `<field>:<value>` expressions to select all documents that contain the `<field>` with the specified `<value>`:
  - `db.ActiveBookings.find( { city: "Torino" } )`

We are interested in object with `key:value` as above
Thus - we need to use the `{ }` to state the we filter on those `keys` whose `value` is `"Torino"`

Conditions

- You can specify query **filters** or criteria that identify the documents to return
- A query filter document can specify **equality** condition with `<field>:<value>` expressions to select all documents that (i) contain the `<field>` (ii) with the specified `<value>`:
  - `db.ActiveBookings.find( { city: "Torino" } )`
- We can form boolean expressions with **AND operator**
  - `db.ActiveBookings.find( {city: "Torino" , interior: "GOOD"} )`



The **,** **(comma)** combines two expressions forming an **$and** operator.
Note: it is a **SINGLE** **`{ <query> }`** statement!

mongoDB.

Conditions – $or operator

- The **$or** operator performs a logical OR operation on an array of *two or more* <expressions> and selects the documents that satisfy *at least* one of the <expressions>
  - can be expressed as a set

{ $or: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

```
db.ActiveBookings.find(
        {   $or: [
                {city: "Torino"} , {city: "Milano"}
            ]
        }
).pretty()
```

Conditions – $or operator

- The **$or** operator can be expressed as a set

```
db.ActiveBookings.find(
     {   $or: [
              {city: "Torino"} , {city: "Milano"}
          ]
     }
).pretty()
```

We use an **operator**
whose **arguments** are specified after the semicolon **:**

Conditions – $or operator

- The **$or** operator can be expressed as a set

```
db.ActiveBookings.find(
     {   $or: [
              {city: "Torino"} , {city: "Milano"}
          ]
     }
).pretty()
```

For the **$or** operator,
the **argument** is an **array** of
`{ expressions }`

Conditions - $and operator

- The **$and** operator performs a logical AND operation on an array of *two or more* <expressions> and selects the documents that satisfy *all* of the <expressions>

{ $and: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

- MongoDB provides an implicit AND operation when specifying a comma separated list of expressions

```
db.ActiveBookings.find( {city: "Torino", interior: "GOOD"}).count()
```

Is equivalent to

```
db.ActiveBookings.find( { $and:
    [
        {city: "Torino"},
        {interior: "GOOD"}
    ]
}).count()
```

Conditions: combining $and and $or

- The **$and** and **$or** operators can be combined into a complicated check

```
db.ActiveBookings.find( {
        interior: "GOOD"
        ,       // AND
        $or: [
                {city: "Torino"} , {city: "Milano"}
        ]
    }
).count()
```

```
db.ActiveBookings.find( { $and: [
        { interior: "GOOD"},
        { $or: [ {city: "Torino"} , {city: "Milano"}]}
        ]
    }
).count()
```

A single query {}
With two checks in AND
The second is in $or

mongoDB.

Query on embedded documents

- When the field holds an **embedded document**, a query can
  - specify an exact match on the **entire** embedded document
  - or specify a match by individual fields in the embedded document using the dot notation: **`object.element.innerElement`**

```
db.ActiveBookings.find( {"driving.distance": -1 } ).pretty()
```

Get the element "**distance**"
of the embedded object "**driving**"
Note: must be enclosed in ""

Operators

Operation	Syntax	Example
Equality	`{ <key>:<value>}`	`{city : "Torino}`
Less than	`{ <key>: { $lt: <value>} }`	`{ init_fuel: { $lt: 4} }`
Less than or equal	`{ <key>: { $lte: <value>} }`	`{ init_fuel: { $lte: 3} }`
Greater than	`{ <key>: { $gt: <value>} }`	`{ init_fuel: { $gt: 4} }`
Greater than or equal	`{ <key>: { $gte: <value>} }`	`{ init_fuel: { $gte: 5} }`
Not equal	`{ <key>: { $ne: <value>} }`	`{ init_fuel: { $ne: 100} }`

The `<key>` is compared against an **expression**, which is expressed as a `{key:value}` element

Same here: we have a `{<query>}`, which contains an expression

mongoDB.

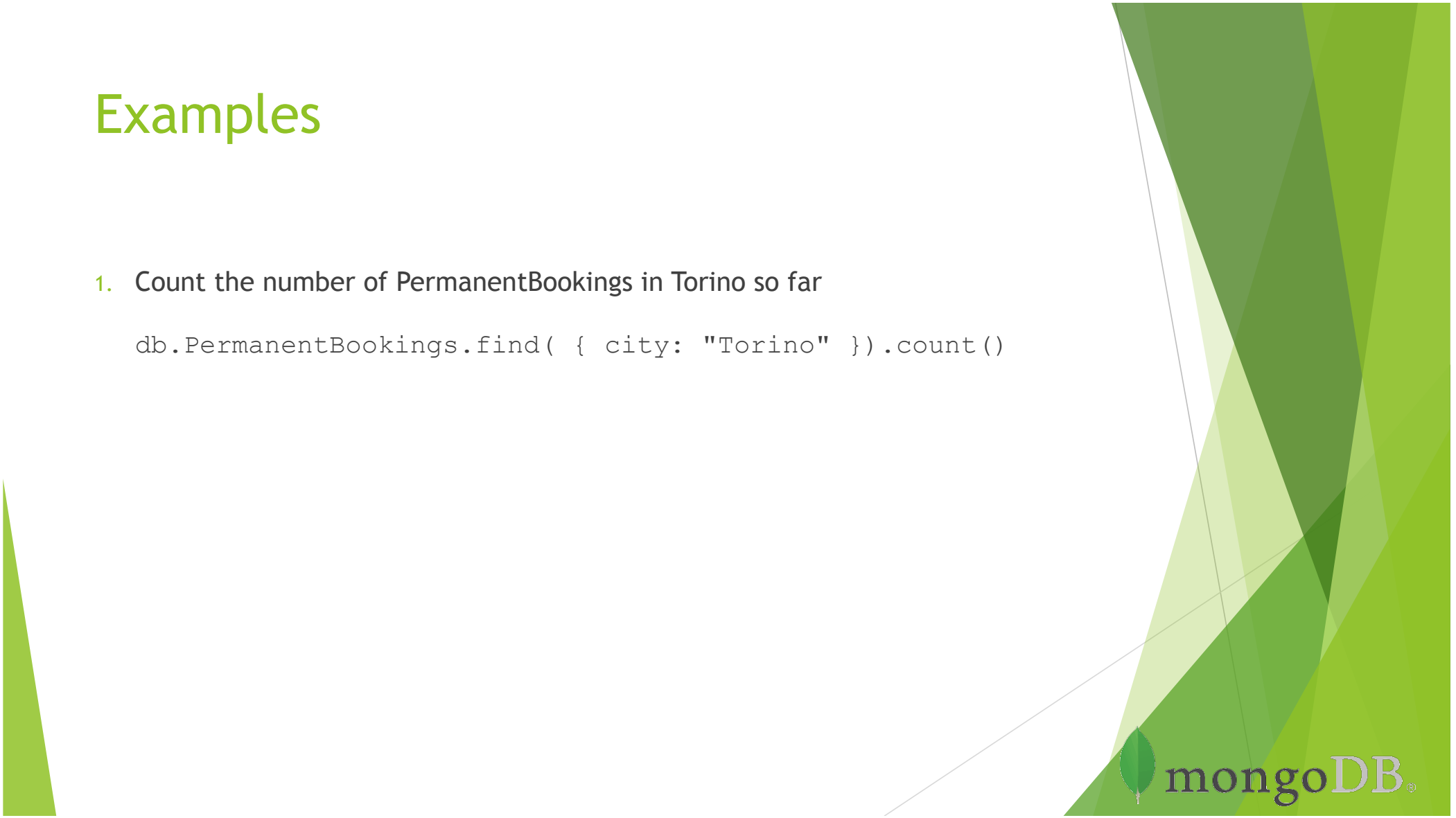
Examples

1. Count the number of PermanentBookings in Torino so far
2. Count the number of PermanentBookings in Torino so far which have also driving time as returned by google map
   - Remember: the system queries google to get that… but google limits the query per days so not all bookings get that

Examples

1. Count the number of PermanentBookings in Torino so far

```
db.PermanentBookings.find( { city: "Torino" }).count()
```



Examples

1. Count the number of PermanentBookings in Torino so far
2. Count the number of PermanentBookings in Torino so far which have also driving time as returned by google

```
db.getCollection('PermanentBookings').find(
    {
      city: "Torino",
      "driving.duration": {$ne: -1}
    }).count()
```

A single `{ <query> }` expression
With two expressions in `$and`

Examples

- For Torino, Milano and Roma,
  - Count the number of PermanentBookings during the Christmas day
    - Suggestion: google how to convert date to unixtime using JS
    - Suggestion: google how to loop in arrays in JS
- Why those numbers are so close?

Examples

```
var cities = ["Torino", "Milano", "Roma"], len = cities.length
var startUnixTime = new Date("2016-12-25") / 1000
var endUnixTime = new Date("2016-12-26") / 1000
for(i=0; i<len; i++){
 c=cities[i]
 print("Checking " + c + " bookings: " +
  db.PermanentBookings.find({
     city: c,
     init_time: { $gte: startUnixTime, $lte: endUnixTime }
   }).count() +
    " Parkings: " +
   db.PermanentParkings.find({
     city: c,
     init_time: { $gte: startUnixTime, $lte: endUnixTime }
   }).count())
}
```

Check the Date class in JS
It returns the Unixtime in ms
Which timezone???

- Question: How to print the date in Human Readable format?

Projections

Projection

- Projection means selecting only the necessary data rather than selecting whole of the data of a document
  - If a document has 5 fields and you need to show only 3, then select only 3 fields!
- Done simply specifying which fields you want in a query
- **`find()`** method accepts a **second optional parameter** that lists fields that you want to retrieve
  - You need to set a list of fields with value 1 (show) or 0 (hide)
  - NOTE: _id is always shown unless you hide it

```
db.COLLECTION_NAME.find({<query>}, {KEY:1} )
```

Projection

- Example: show `init_time` and `final_time` for ActiveBookings in Torino

```
db.ActiveBookings.find(
 { city: "Torino" } ,                          <query>
 { init_time: 1, final_time: 1, _id: 0}        <projection>
)
```

- Question: why final_time is always set to -1 ?

Sort() method

- The **`sort()`** method accepts a **document** containing a list of fields along with their sorting order
- To specify sorting order
  - 1 is used for ascending order
  - -1 is used for descending order

```
db.COLLECTION_NAME.find().sort( {KEY:1} )
```

- Examples:
  - Get cities with ActiveBookings, and sort by city in descending order

```
db.ActiveBookings.find({}, {city:1}).sort({city: -1})
```

  - Get cities with ActiveBookings, and sort by Init_time in ascending order

```
db.ActiveBookings.find({}, {city:1}).sort({init_time: 1})
```

    - Note: This works because the MongoDB query engine will always apply the sorting first, then the projection later

Sort() method

- You can sort on two or more fields
  - fieldA first, then fieldB second,
  - the mongo JavaScript shell obeys the left-to-right order in the associative array

```
db.myCollection.find().sort( { fieldA: 1, fieldB: 1 } )
```

- Examples:
  - Get cities and init_time with PermanentBookings, and sort by init_time and city

```
db.PermanentBookings.find({}, {city:1, init_time:1, _id:0}  ).sort({init_time:1, city:1})
```

  - Get cities and init_time with PermanentBookings, and sort by city and init_time

```
db.PermanentBookings.find({}, {city:1, init_time:1, _id:0}  ).sort({city:1, init_time:1})
```

- Note: sorting may take long time… you should use indexes to optimize sorting

Examples

- Question: How to print the date in Human Readable format?

```
db.ActiveBookings.find(
        {},
        {city:1, init_time:1, _id:0}
    ).sort(
        {city:1, init_time:1}
    ).forEach(
        function(o){
            var date = new Date(o.init_time*1000)
            print (date + " " + o.city)
        }
    )
```

<query>
<projection>

sorting

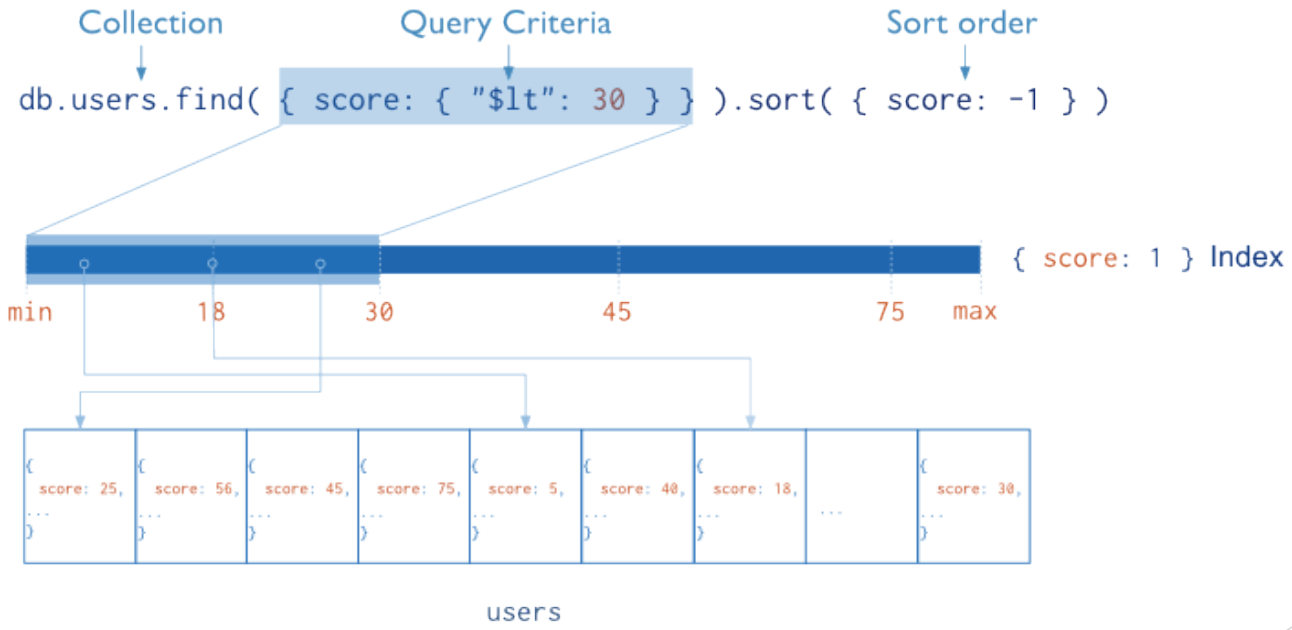
printing (using cursor)

[check the Date class in JS]

Indexes

- Indexes support the efficient resolution of queries
- Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement
  - This scan is highly inefficient and require MongoDB to process a large volume of data
- Indexes are special data structures
  - Store a small portion of the data set in an easy-to-traverse form
  - The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index
  - The ordering of the index entries supports efficient equality matches and range-based query operations
  - MongoDB can return sorted results by using the ordering in the index

Indexes



mongoDB.

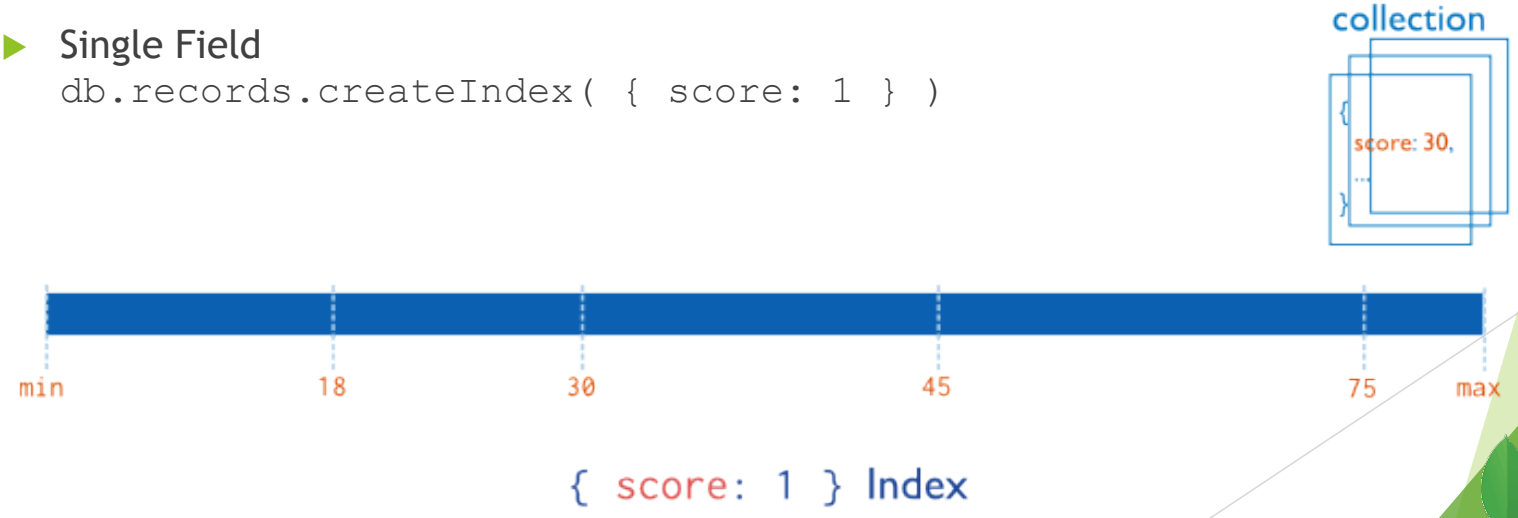
Creating indexes

- To create an index, use **`db.collection.createIndex()`** or a similar method from your driver.

```
db.collection.createIndex( <key and index type specification>, <options> )
```

- MongoDB provides a number of **index types** to support specific types of data and queries
- Single Field

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



mongoDB.

Indexes

- Syntax

```
db.collection.createIndex(keys, options)
```

- Keys: A document that contains the field and value pairs where the field is the index key and the value describes the type of index for that field
- For an ascending index on a field, specify a value of 1; for descending index, specify a value of -1.
- MongoDB supports several different index types including text, geospatial, and hashed indexes.

Indexes

- Examples:
  - Which indexes are present in the PermanentBookings collection
  - Check the speed of counting
    - How many bookings have been so far in Torino
    - How many bookings have the google map information
    - How many bookings in Torino have the google map information
  - Note: how does MongDB run the latter query? Why the second one take so much more to execute?

Aggregations

Aggregations

- Aggregations 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:
  - **Single purpose aggregation methods**
  - **Aggregation pipeline**
  - Map-reduce function

Single purpose aggregation operations

- MongoDB provides simple aggregation operations: `.count()` and `.distinct()`
- These operations aggregate documents from a single collection
  - While these operations provide simple access to common aggregation processes, they lack the flexibility and capabilities of the aggregation pipeline and map-reduce

Single purpose aggregation operations

- MongoDB provides simple aggregation operations: **`.count()`**

`db.collection.count( query, options)`

Count all documents in the collection that satisfy the `query`

```
db.PermanentBookings.count()
db.PermanentBookings.find().count()
db.PermanentBookings.find({city: "Torino"}).count()
db.PermanentBookings.count( {city: "Torino"})
db.PermanentBookings.count( {city: {$eq: "Torino"} })
```

note: `.find()` and expressions returns a collection… so that you can `.count()` elements…

Single purpose aggregation operations

- MongoDB provides simple aggregation operations: **`.distinct()`**

`db.collection.distinct( field, query, options)`

Finds the distinct values for a specified `field` across a single collection and returns the results in an array

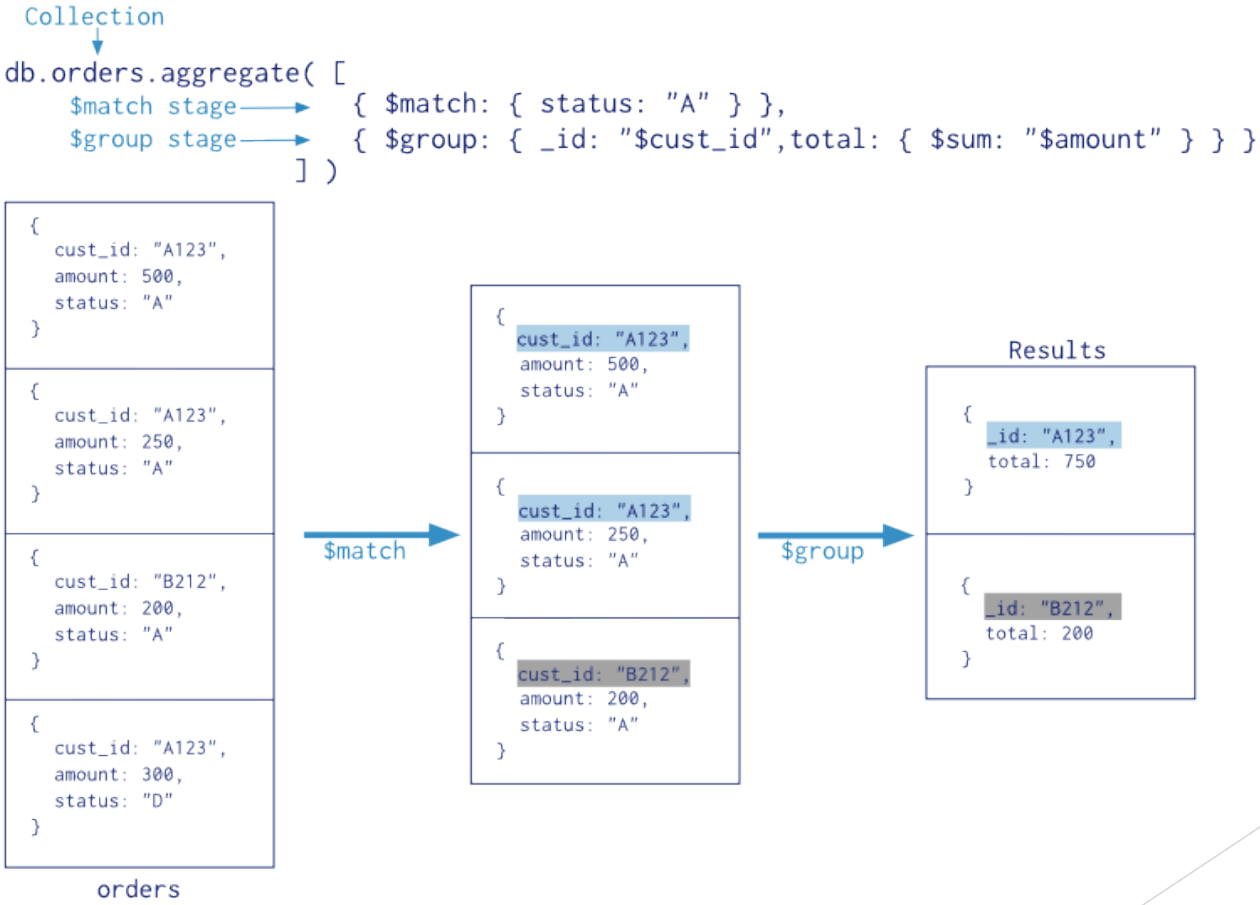
`db.ActiveBookings.distinct("city")`: returns the different values taken by `city`

```
db.ActiveBookings.distinct("city", {city: "Torino"})
db.ActiveBookings.distinct("plate", {city: "Torino"})
db.PermanentBookings.distinct( "plate",{$or: [{city:  "Torino"}, {city: "Milano"}]})
db.PermanentBookings.distinct( "plate",{$and: [{city:  "Torino"}, {city: "Milano"}]})
```

Aggregation pipeline

- MongoDB's aggregation framework is modelled on the concept of data processing pipelines
  - Documents enter a multi-stage pipeline that transforms the documents into an aggregated result
- The most basic pipeline stages provide ***filters*** that operate like queries and ***document transformations*** that modify the form of the output document
- Other pipeline operations provide tools for **grouping** and **sorting** documents by specific field or fields
- Pipeline stages can use operators for tasks such as calculating the average or concatenating a string
- The pipeline provides efficient data aggregation using native operations within MongoDB, and is the preferred method for data aggregation in MongoDB

Aggregation pipeline



mongoDB.

Aggregation pipeline

- The MongoDB aggregation **`pipeline`** consists of stages

  `db.collection.aggregate( [ pipeline, option ] )`
- Each stage transforms the documents as they pass through the pipeline
  - Pipeline stages do not need to produce one output document for every input document; e.g., some stages may generate new documents, or filter out documents
  - Pipeline stages can appear multiple times in the pipeline
- **Operators** define the transformation of a stage
  - **$project:** Used to **select** some specific fields from a collection
  - **$match:** This is a **filtering** operation and thus this can reduce the amount of documents that are given as input to the next stage
  - **$group:** This does the actual **aggregation**
  - **$sort: Sorts** the documents
  - **$limit:** This limits the amount of documents to look at, by the given number starting from the current positions
  - **$out:** Takes the documents returned by the aggregation pipeline and writes them to a specified collection. Must be the last stage in the pipeline.

$project stage in aggregations

Aggregation: $project

`{ $project: { <specification> } }`

- Passes along the documents with only the specified fields to the next stage in the pipeline
- The specified fields can be **existing fields** from the input documents

`{ fieldA: 1, fieldB: 1, …, _id: 0 }`

```
db.ActiveBookings.aggregate([
    {$project: { city: 1, _id:0 } }
])
```

A single stage pipeline which is a `$project` operation

is equivalent to

```
db.ActiveBookings.find( {}, { city: 1, _id: 0} } )
```

- If you specify an inclusion of a field that does not exist in the document, $project ignores that field inclusion

mongoDB.

Aggregation: $project

- The specified fields can be **newly computed fields**
  - To add a new field or to reset the value of an existing field, specify the field name and set its value to some expression

```
{ $project: { newfield: { <expression>} } }
```

- **Expressions** can include
  - **field paths** to access fields in the input document
    - prefix with a dollar sign $ the field name or the dotted field name

```
copyOfCity: "$city"
```

  - **Operator expression**
    - Operator expressions are similar to functions that take arguments
    - Take an array of arguments with the following form:

```
{ <operator>: [ <argument1>, <argument2> ... ] }
```

Boolean and comparison expressions

- **Boolean expressions**
  - `$and, $or`: Returns true only when *all* or *any* its expressions evaluate to true. Accepts any number of argument expressions
  - `$not`: Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression
- **Comparison expressions**
  - `$cmp`: Returns: 0 if the two values are equivalent, 1 if the first value is greater than the second, and -1 if the first value is less than the second
  - `$eq, $gt, $gte, $lt, $lte, $ne`: Return true if the values are equivalent, greater than, greater or equal to, less than, less or equal to, not equal
- Example

```
test: { $and: [ { $eq: ["$city", "Stuttgart"]}, {$eq: ["$exterior", "GOOD"] }] }
```

Arithmetic operators

- Arithmetic expressions perform mathematic operations on numbers
  - Some arithmetic expressions can also support date arithmetic

Name	Description
`$add`	Adds numbers to return the sum, or adds numbers and a date to return a new date
`$divide`	Returns the result of dividing the first number by the second
`$multiply`	Multiplies numbers to return the product. Accepts any number of argument expressions
`$subtract`	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds

Example

- Compute and return the duration of rentals

```
db.PermanentBookings.aggregate( [
{
    $project:
    {
        duration: {
            $subtract: ["$final_time", "$init_time”]
            }
    }
}
] )
```

Example

- Compute and return the duration of rentals

```
db.PermanentBookings.aggregate( [
{
    $project:
    {
        duration: {
            $subtract: ["$final_time", "$init_time”]
            }
    }
}
] )
```

A single stage aggregation

Example

- Compute and return the duration of rentals

```
db.PermanentBookings.aggregate( [
{
    $project:
    {
        duration: {
            $subtract: ["$final_time", "$init_time”]
            }
    }
}
] )
```

A projection stage

Example

- Compute and return the duration of rentals

```
db.PermanentBookings.aggregate( [
{
    $project:
    {
        duration: {
            $subtract: ["$final_time", "$init_time”]
            }
    }
}
] )
```

Computing an expression
Whose key would be
`duration`
[plus the default `_id`]

Example

- Compute and return the duration of rentals

```
db.PermanentBookings.aggregate( [
{
    $project:
    {
        duration: {
            $subtract: ["$final_time", "$init_time"]
            }
    }
}
] )
```

Computed as a `$subtract` expression between two elements

Expressions

- There are lot of possible expressions other than boolean, comparison and math
  - Set expressions: performs set operation on arrays, treating arrays as sets.
  - String expressions: `$concat, $split, $toLower, $toUpper, …`
  - Array expressions: `$isArray, $range, $size, …`
  - Date expressions: `$dayOfYear, $DayOfMonth, $DayOfWeek`

- See https://docs.mongodb.com/manual/meta/aggregation-quick-reference/#aggregation-expressions for details

$match stage in aggregations

$match

```
{ $match: { <query> } }
```

- **`$match`** filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage
- **`$match`** uses standard MongoDB queries
  - For each input document, outputs either one document (a match) or zero documents (no match)
- Pipeline Optimization
  - Place the `$match` as early in the aggregation pipeline as possible. Because `$match` limits the total number of documents in the aggregation pipeline, earlier `$match` operations minimize the amount of processing down the pipe.
  - If you place a `$match` at the very beginning of a pipeline, the query can take advantage of indexes like any other `db.collection.find()`

Example

- Compute and return the duration of rentals in Torino only

```
db.PermanentBookings.aggregate([
    { $match: { city: "Torino"} },
    { $project: {
       _id: 0,
       city: 1,
       init_time: 1,
       duration: {$subtract: ["$final_time", "$init_time"] }
       }
    }
])
```

Example

- Compute and return the duration of rentals in Torino only

```
db.PermanentBookings.aggregate([
    { $match: { city: "Torino"} },
    { $project: {
       _id: 0,
       city: 1,
       init_time: 1,
       duration: {$subtract: ["$final_time", "$init_time"] }
       }
    }
])
```

`$match` stage

Example

- Compute and return the duration of rentals in Torino only

```
db.PermanentBookings.aggregate([
    { $match: { city: "Torino"} },
    { $project: {
       _id: 0,
       city: 1,
       init_time: 1,
       duration: {$subtract: ["$final_time", "$init_time"] }
       }
    }
])
```

$match stage

$project stage

$group stage in aggregations

$group stage

```
{ $group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... } }
```

- Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping
- The output documents contain an `_id` field which contains the distinct group by key
- The output documents can also contain computed fields that hold the values of some accumulator expression grouped by the `$group`'s `_id` field
- $group does *not* order its output documents
- The `_id` field is *mandatory*
- The remaining computed fields are *optional* and computed using the `<accumulator>` operators
- Note: the `$group` stage has a limit of 100 megabytes of RAM

$group example

- Count the number of ActiveBooking in each city

```
db.ActiveBookings.aggregate([
    {
        $group : {
            _id: "$city",
            count: {$sum: 1}
            }
    }
])
```

$group example

- Count the number of ActiveBooking in each city

```
db.ActiveBookings.aggregate([
    {
        $group : {
            _id: "$city",
            count: {$sum: 1}
            }
    }
])
```

A single stage aggregation

$group example

- Count the number of ActiveBooking in each city

```
db.ActiveBookings.aggregate([
    {
        $group : {
            _id: "$city",
            count: {$sum: 1}
            }
    }
] )
```

Group using the "$city" as key

And then $sum the number of matches in count

Accumulator Operations

Name	Description
$sum	Returns a sum of numerical values. Ignores non-numeric values
$avg	Returns an average of numerical values. Ignores non-numeric values
$first	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order
$last	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order
$max	Returns the highest expression value for each group
$min	Returns the lowest expression value for each group
$push	Returns an array of expression values for each group
$addToSet	Returns an array of unique expression values for each group. Order of the array elements is undefined
$stdDevPop	Returns the population standard deviation of the input values
$stdDevSamp	Returns the sample standard deviation of the input values

$sort stage in aggregation

$sort stage

```
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
```

- Sorts all input documents and returns them to the pipeline in sorted order
- **`$sort`** takes a document that specifies the field(s) to sort by and the respective sort order. **`<sort order>`** can have one of the following values:
  - **`1`** to specify ascending order
  - **`-1`** to specify descending order
- Note: The $sort stage has a limit of 100 megabytes of RAM

$sort example

- Count the number of ActiveBooking in each city and sort results

```
db.ActiveBookings.aggregate([
    {
        $group : {
            _id: "$city",
            count: {$sum: 1}
            }
    },
    {
        $sort : { count: 1}
    }
] )
```

Putting everything together

- Considering the Christmas day in 2016
- For each city, compute
  - The number of rentals
  - The total rental time
  - The average rental duration
  - The total rental revenue (assuming a 25c/min rate)
  - The average rental cost
- Sort results per increasing number of rentals
- Note: consider only possible actual rentals
  - Whose initial and final position differ
  - Whose duration is “reasonable”

```
var startUnixTime = new Date("2016-12-29") / 1000 - 3600
var endUnixTime = new Date("2016-12-30") / 1000 - 3600
db.PermanentBookings.aggregate([
    { $match: { // Get only those rental in the selected period
         $and: [
            {init_time: { $gte: startUnixTime}},
            {init_time: { $lte: endUnixTime }} ]
        }
    },
    { $project: { // compute rental duration, and distance traveled
        city : 1,
        distance_lat: { $subtract: ["$init_lat","$final_lat" ] },
        distance_lon: { $subtract: ["$init_lon","$final_lon" ] },
        duration: {$divide: [{$subtract: ["$final_time", "$init_time"] }, 60]},
        }
    },
    { $match: { // check that the car was moved
          $and: [
            {distance_lat: { $gt: 0}},
             {distance_lon: {$gt: 0}},
             {duration: {$gt: 1, $lt: 120}} ]
        }
    },
    { $project: { // get then the possible cost for this rental
        city : 1,
        duration: 1,
        distance: {$add: ["$distance_lat","$distance_lon"]},
        cost: {$multiply: ["$duration", 0.25]}
        }
    },
    { $group: { // now compute the totals, per city
        _id: "$city",
        tot_rentals: {$sum: 1},
        tot_time: {$sum: "$duration"},
        avg_time: {$avg: "$duration"},
        tot_cost: {$sum: "$cost"},
        avg_cost: {$avg: "$cost"}
        }
    },
    { $sort: { // last stage -- sort by tot_rentals
        tot_rentals: 1
        }
    }
])
```

Match the period

Get the needed fields

Match likely rentals

Get the interesting fields

Group by city
And compute statistics

Sort by number of rentals

mongoDB.