TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

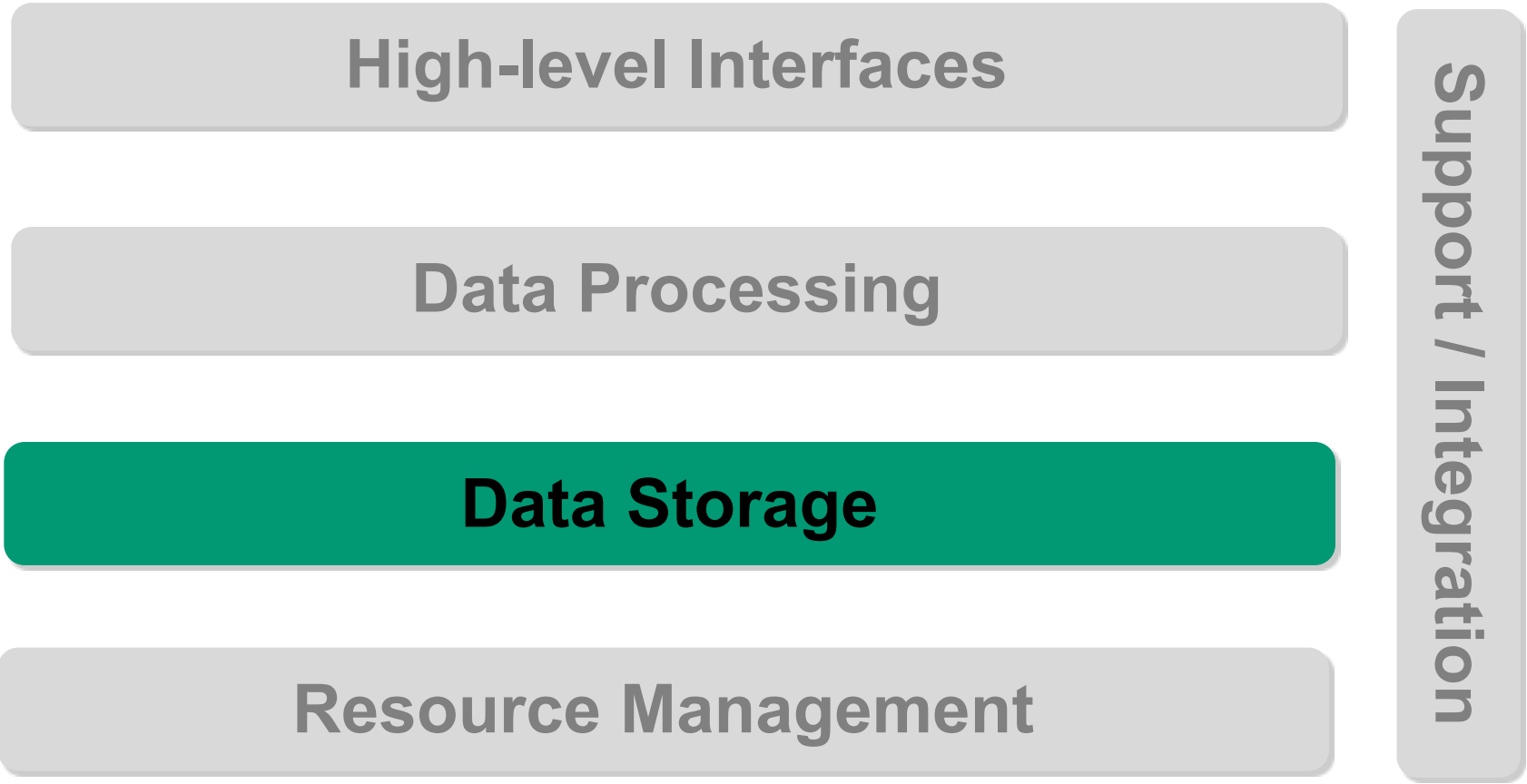
NoSQL: Redis and MongoDB

A.A. 2022/23

Matteo Nardelli

Laurea Magistrale in
Ingegneria Informatica - II anno

The reference Big Data stack



NoSQL data stores

Main features of NoSQL (**Not Only SQL**) data stores:

- Support flexible schema
- Scale horizontally
- Provide scalability and high availability by storing and replicating data in distributed systems
- Do not typically support ACID properties, but rather BASE

Simple APIs

- Low-level data manipulation and selection methods
- Queries capabilities are often limited

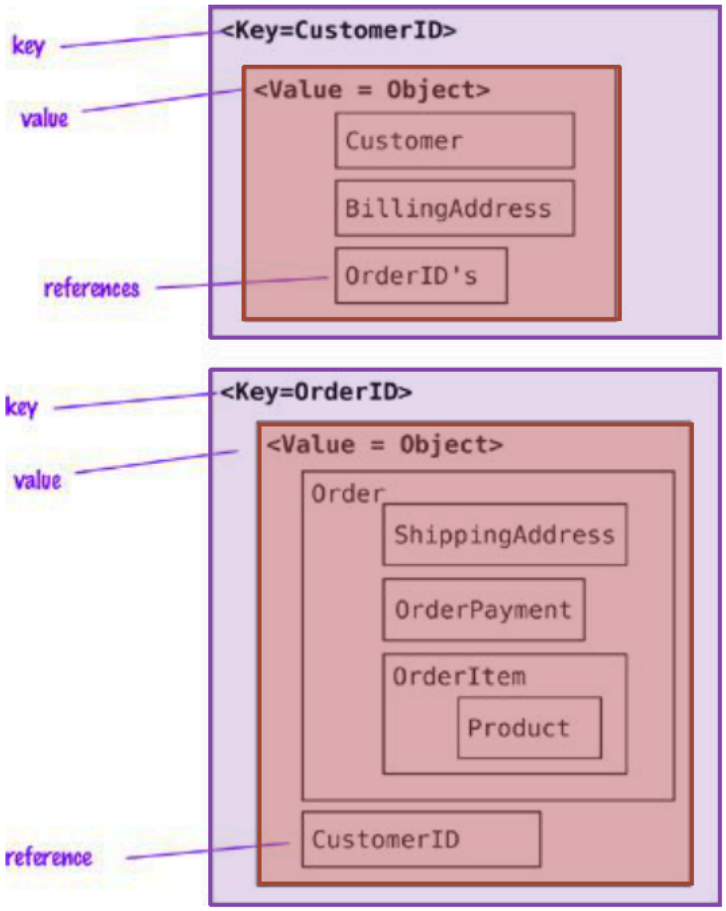
Data models for NoSQL systems:

- Aggregate-oriented models:
  key-value, **document**, and **column-family**
- **Graph-based** models

Key-value data model

- Simple data model:
  - data as a collection of key-value pairs
- Strongly aggregate-oriented
  - A set of <key,value> pairs
  - Value: an aggregate instance
  - A value is mapped to a **unique** key
- The aggregate is opaque to the database
  - Values do not have a known structure
  - Just a big blob of mostly meaningless bit
- Access to an aggregate:
  - Lookup based on its key
- Richer data models can be implemented on top

Key-value data model: example



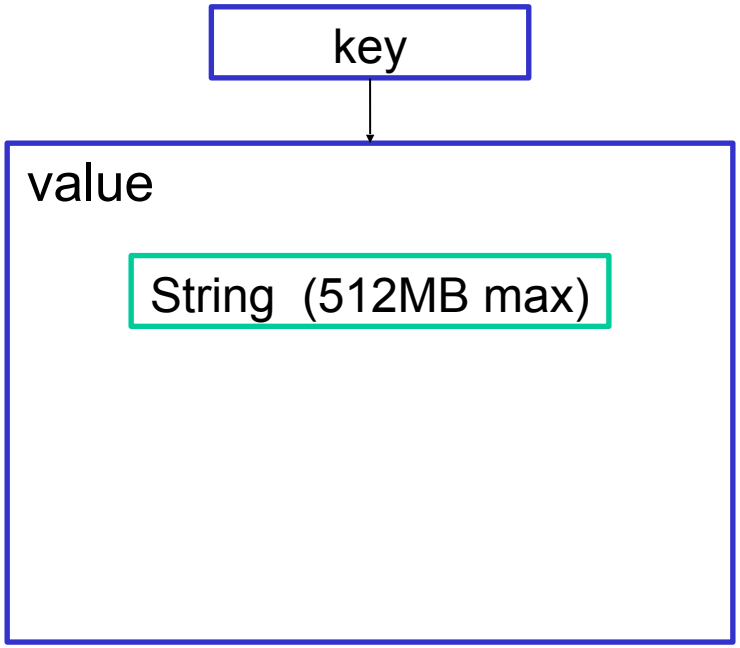
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: **Strings**
  - Containers (of strings):
    - Hashes
    - Lists
    - Sets
    - Sorted Sets



https://redis.io/topics/data-types

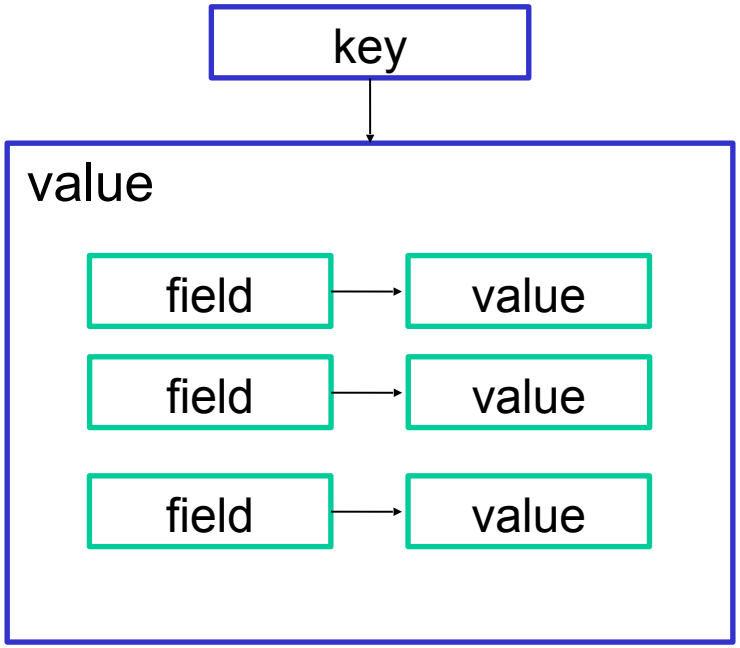
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: Strings
  - Containers (of strings):
    - **Hashes**
    - Lists
    - Sets
    - Sorted Sets



https://redis.io/topics/data-types

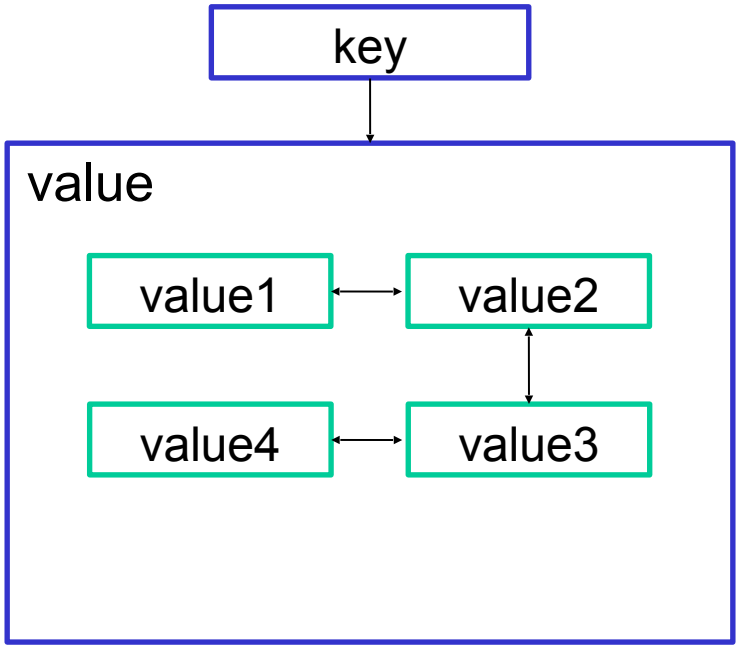
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: Strings
  - Containers (of strings):
    - Hashes
    - **Lists**
    - Sets
    - Sorted Sets



https://redis.io/topics/data-types

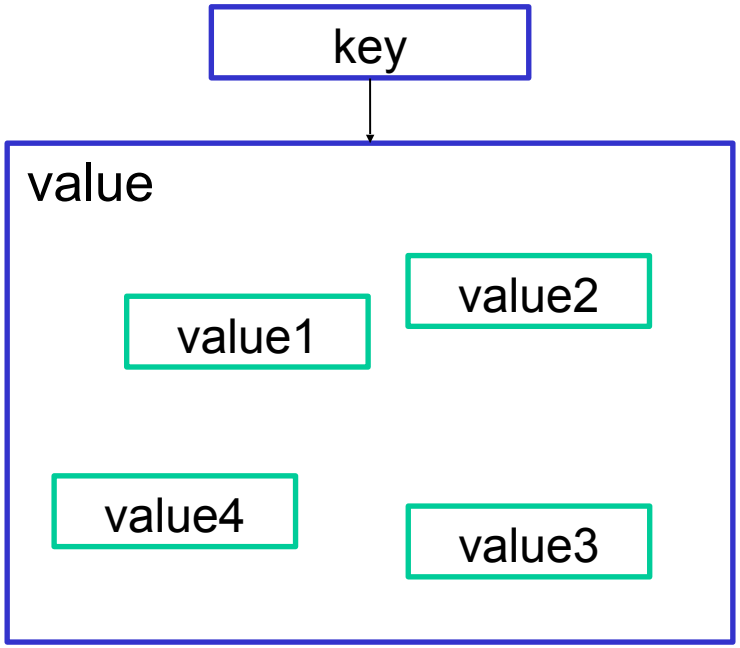
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: Strings
  - Containers (of strings):
    - Hashes
    - Lists
    - **Sets**
    - Sorted Sets



https://redis.io/topics/data-types

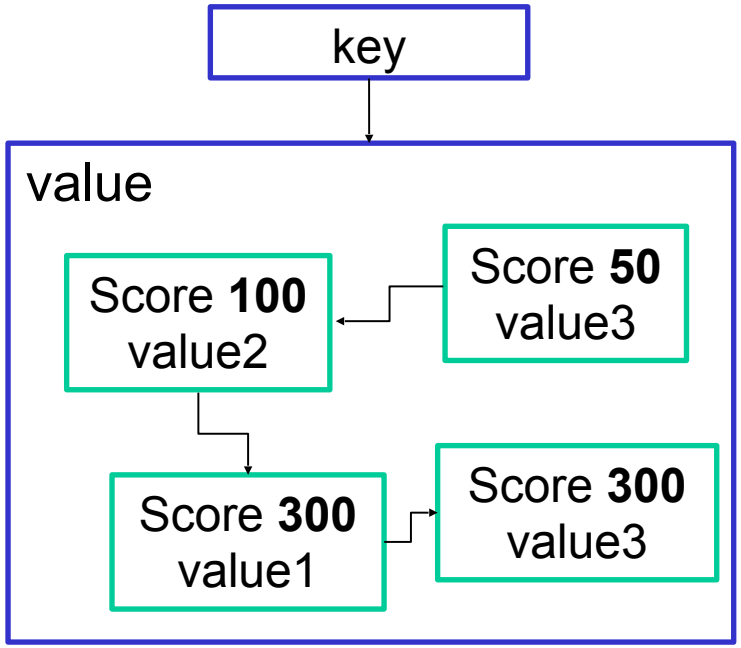
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: Strings
  - Containers (of strings):
    - Hashes
    - Lists
    - Sets
    - **Sorted Sets**



https://redis.io/topics/data-types

Hands-on Redis
(Docker image)

Redis with Dockers

- We use a lightweight container with redis preconfigured

```
$ docker pull sickp/alpine-redis
```

- create a small network named `redis_network` with one redis server and one client

```
$ docker network create redis_network


$ docker run --rm --network=redis_network --
name=redis-server sickp/alpine-redis


$ docker run --rm --net=redis_network -it sickp/
alpine-redis redis-cli -h redis-server
```

Redis with Dockers

- Use the command line interface on the client to connect to the redis server

```
$  redis-cli -h redis-server [-p (port-number)]
```

Atomic Operations: Strings

Main operations, implemented in an atomic manner:

```
redis> GET key
redis> SET key value [EX expiration-period-secs]
redis> APPEND key value
redis> EXISTS key
redis> DEL key
redis> KEYS pattern   # use SCAN in production
```

```
# set if key does not exist
redis> SETNX key value
# Get old value and set a new one
redis> GETSET key value
# Set a timeout after which the key will be deleted
redis> EXPIRE key seconds
```

Details on Redis commands: https://redis.io/commands/

Atomic Operations: Hashes

Main operations, implemented in an atomic manner:

```
redis> HGET key field
redis> HSET key field value
redis> HEXISTS key field
redis> HDEL key field
```

```
# Get all field names of the hash stored at key
redis> HKEYS key
# Get all values of the hash stored at key
redis> HVALS key
```

Details on Redis commands: https://redis.io/commands/

Case Study (1)

- **Problem:** We need to implement a recommendation system for dynamically propose a radio station, that suggests the next song according to the history of played songs per genre.

Case Study (2)

- **Problem:** We need to implement a recommendation system for dynamically propose a radio station, that suggests the next song according to the history of played songs per genre.

- **Solution:** we need to keep trace of a counter for each genre played by the user. We consider to store a userXcounter for each user and, for each userXcounter, we use an hashmap that associates a counter to each genre played.

Case Study (3)

```
redis> HSET user1counter rock 1
redis> HGET user1counter rock
redis> HEXISTS user1counter classic
redis> HGET user1counter classic
redis> HSET user1counter rock 4
redis> HGET usr1counter rock
redis> HSET user1counter jazz 2
redis> HSET user1counter pop 1
redis> HEXISTS user1counter classic
redis> HDEL user1counter classic
redis> HEXISTS user1counter classic
```

Case Study (4)

```
redis> HKEYS user1counter
               1) “rock”
               2) “jazz”
               3) “pop”
redis> HVALS user1counter
               1) “4”
               2) “2”
               3) “1”
```

Atomic Operations: Sets

Main operations, implemented in an atomic manner:

```
# Add a value to the set stored at key
redis> SADD key value
# Remove the value from the set stored at key
redis> SREM key value
# Get the cardinality of the set stored at key
redis> SCARD key
# Remove and return a random member of the set
redis> SPOP key
```

```
# Union, Difference, Intersection between sets
redis> SUNION keyA keyB
redis> SDIFF keyA keyB
redis> SINTER keyA keyB
```

Details on Redis commands: https://redis.io/commands/

Case Study (5)

- Problem: We also need to retrieve bands or singers that play that musical genre. We can rely on the data store to memorize bands/ singers per each musical genre. We assume that a band can play several genres. We might be interested in selecting bands belonging to multiple genres, or in identifying a selection of bands that play the same kind of music.

Case Study (6)

- Problem: We also need to retrieve bands or singers that play that musical genre. We can rely on the data store to memorize bands/ singers per each musical genre. We assume that a band can play several genres. We might be interested in selecting bands belonging to multiple genres, or in identifying a selection of bands that play the same kind of music.

- Solution: we need to keep trace of a set of singers for each musical genre.

Case Study (7)

```
redis> SADD rock "pink floyd"
redis> SADD rock "queen"
redis> SADD rock "nirvana"
redis> SADD rock "baustelle"
redis> SADD jazz "paolo conte"
redis> SADD pop "paolo conte"
redis> SADD pop "baustelle"
redis> SCARD rock                    # 4
redis> SCARD Rock                    # 0
redis> SADD pop "mozart"
redis> SREM pop "mozart"
```

Case Study (8)

```
redis> SDIFF rock pop
                    1) “pink floyd”
                    2) “queen”
                    3) “nirvana”
redis> SUNION rock jazz
                    1) “pink floyd”
                    2) “queen”
                    3) “nirvana”
                    4) “baustelle”
                    5) “paolo conte”
```

Case Study (9)

- **Problem:** The recommendation system might learn from the user behavior upon the suggested songs. Therefore, we need to identify the number of reproduction of the suggested genres, so that, in the future, we can suggest the top-K genres that have been listened by the user.

Case Study (10)

- **Problem:** The recommendation system might learn from the user behavior upon the suggested songs. Therefore, we need to identify the number of reproduction of the suggested genres, so that, in the future, we can suggest the top-K genres that have been listened by the user.

- **Solution:** We can use the sorted sets to store the number of reproduced songs by genre, and let the data structure automatically determines the top-K elements.

Atomic Operations: Sorted Sets

Sorted Sets: non repeating collections of strings.

A score is associated to each value. Values of a set are ordered, from the smallest to the greatest score. Scores may be repeated.

Main operations, implemented in an atomic manner:

```
# Add a value to the set stored at key
redis> ZADD key score value
# Remove the value from the set stored at key
redis> ZREM key value
# Get the cardinality of the set stored at key
redis> ZCARD key
# Return the score of a value in the set stored at key
redis> ZSCORE key value
```

Details on Redis commands: https://redis.io/commands/

Atomic Operations: Sorted Sets

```
redis> ZCARD urepr
redis> ZADD urepr 1 rock
redis> ZADD urepr 1 jazz
redis> ZADD urepr 1 pop
redis> ZCARD urepr                    # 3
redis> ZREM urepr pop
redis> ZCARD urepr                    # 2
redis> ZSCORE urepr jazz              # 1
```

Atomic Operations: Sorted Sets

The presence of a score enables to rank or to retrieve the elements as well as changing their order during the lifetime of the sorted set

```
# Returns the rank of value in the sorted set.
# The rank is 0-based.
redis> ZRANK key value

# Returns the values in a range of the ranking (start and
stop are 0-based indexes; -k stands for the k element from
the end of the rank)
redis> ZRANGE key start stop [WITHSCORES]
# Like ZRANGE but uses the score instead of the index
redis> ZRANGEBYSCORE key min max

# Increments by increment the score of value
redis> ZINCRBY key increment value
```

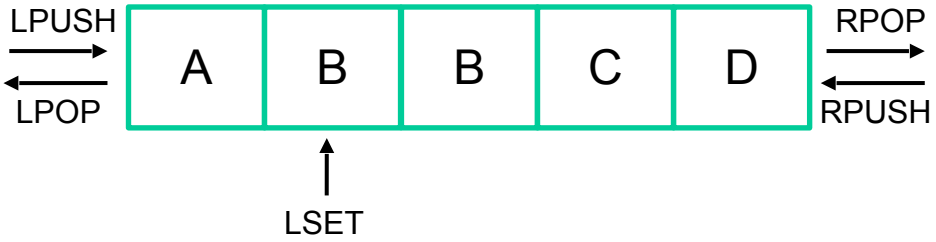
Details on Redis commands: https://redis.io/commands/

Case Study (12)

```
redis> ZRANK urepr pop
redis> ZRANK urepr rock                    # 1
redis> ZINCRBY urepr 3 rock                # score:4
redis> ZINCRBY urepr 1 pop                 # score:1
redis> ZCARD urepr                         # 3
redis> ZRANK urepr pop                     # 1
redis> ZRANK urepr rock                    # 2
redis> ZRANGE urepr 0 1
                      1) “jazz”
                      2) “pop”
redis> ZRANGE urepr 0 -1
                      1) “jazz”
                      2) “pop”
                      3) “rock”
```

Atomic Operations: Lists

Lists are ordinary linked lists; they enable to push and pop values at both sides or in an exact position



Main operations, implemented in an atomic manner:

```
# Push value at the head|tail of the list in key
redis> LPUSH|RPUSH key value [value]
# Remove and return the head|tail of the list in key
redis> LPOP|RPOP key
# Get the length of the list
redis> LLEN key
# Returns the specified elements of the list (0-based) index
redis> LRANGE key start stop
```

Case Study (13)

The only music player needs to store the playlist for the user, which can be populated by the user or by the recommendation system.

```
redis> RPUSH uplay "time"
redis> RPUSH uplay "money"
redis> LPUSH uplay "glory days"
redis> LLEN uplay                                  # 3
redis> LRANGE uplay 0 -1

                    1) “glory days”
                    2) “time”
                    3) “money”
redis> LRANGE uplay -2 -1
                    1) “time”
                    2) “money”
```

Atomic Operations: Lists

```
# Removes the first count occurrences of elements equal to
value from the list stored at key

redis> LREM key count value
```

count > 0	remove elements equal to value moving from head to tail
count < 0	remove elements equal to value moving from tail to head
count = 0	remove all elements equal to value.

```
# Sets the list element at (0-based) index to value.

redis> LSET key index value
```

Details on Redis commands: https://redis.io/commands/

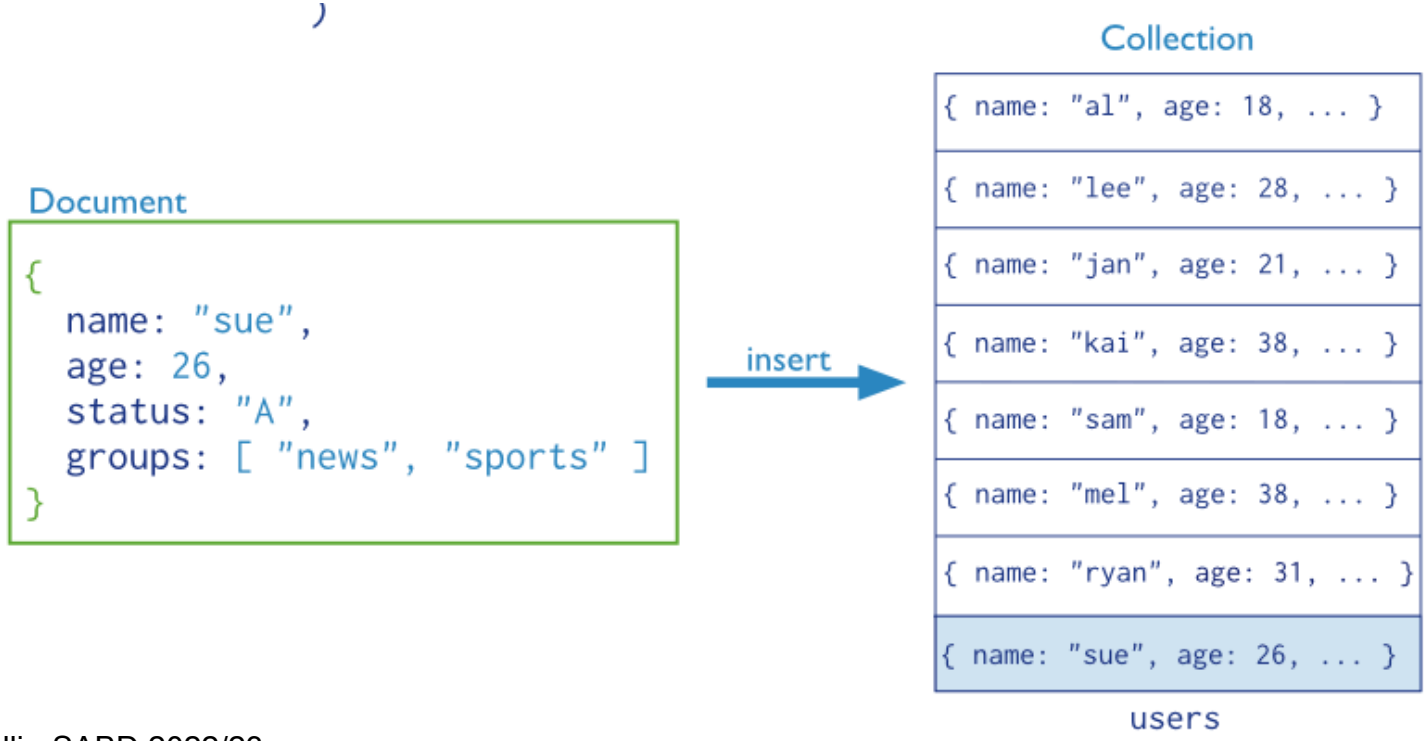
Document data model

Document store: derived from the key-value data model

- Data model:
  - A set of <key,document> pairs
  - Document: an aggregate instance
- A document:
  - can contain complex data structures (nested objects)
  - does not require adherence to a fixed schema
- Access to the aggregate (document):
  - Structure of the aggregate visible
    - Often there are limitations on its content type
  - Queries based on the fields in the aggregate

In MongoDB:

- documents are grouped together into collections;
- inside each collection, a document should have a unique key;
- Documents can have different schema.



MongoDB

RDMS (e.g., mysql)	MongoDB
Tables	Collections
Records/Rows	Documents
Queries return record(s)	Queries return a cursor

Document

```
{
  name: "sue",          <-- field: value
  age: 26,              <-- field: value
  status: "A",          <-- field: value
  groups: [ "news", "sports" ]   <-- field: value
}
```

MongoDB

MongoDB represents JSON documents using BSON, a binary-encoded format that extends the JSON model to provide additional data types.

Data Types

- String: combination of characters
- Boolean: True or False
- Integer: digits
- Double: a type of floating point number
- Null: not zero, not empty
- Array: a list of values
- Object: an entity which can be used in programming (value, variable, function, or data structure).
- Timestamp: a 64 bit value referring to a time
- Internationalized Strings: UTF-8 for strings
- Object IDs: every document must have an Object ID which is unique

An example of document structure

```
{
 _id: ObjectId("5099803df3f4948bd2f98391"),
 name: { first: "Alan", last: "Turing" },
 birth: new Date('Jun 23, 1912'),
 death: new Date('Jun 07, 1954'),
 contribs: [ "Turing machine", "Turing test", "Turingery" ],
 views : NumberLong(1250000)
}
```

The above fields have the following data types:

- `_id` holds an ObjectId.
- `name` holds an embedded document that contains first and last.
- `birth` and `death` hold values of the Date type.
- `contribs` holds an array of strings.
- `views` holds a value of the NumberLong type.

Dot notation

MongoDB uses the dot notation to access:

- the elements of an array: by concatenating the array name with the dot (.) and zero-based index position (in quotes)

```
{ ...
  contribs: [ "Turing machine", "Turing test", ... ],
... }
```

e.g., to specify the 3rd element: "contribs.2"

- the fields of an embedded document: by concatenating the embedded document name with the dot (.) and the field name

```
{ ...
  name: { first: "Alan", last: "Turing" },
... }
```

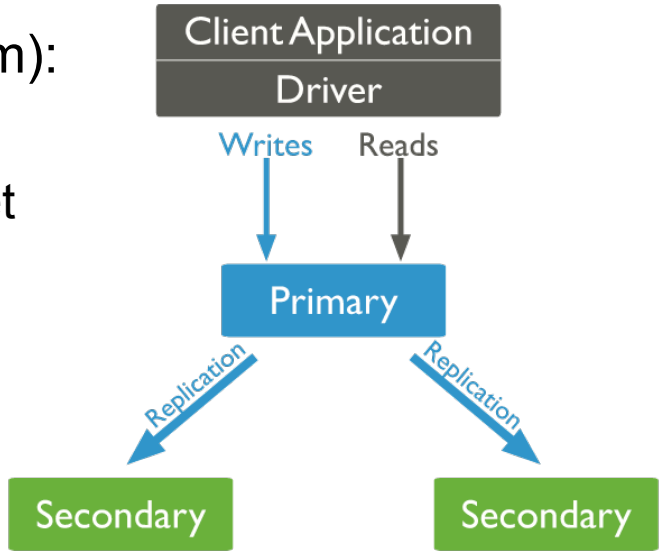
e.g., to specify the last name: "name.last"

https://docs.mongodb.com/manual/core/document/#dot-notation

MongoDB: Consistency and Availability

Consistency (Mongo is considered a CP system):

- Monotonic Writes:
  - For standalone mongo instance and replica set
- Causal Consistency:
  - Across replica sets;
  - Configurable read/write quorum



Atomicity:

- Single document atomicity
- Multi-document Transactions with snapshot isolation

Replication and Scaling:

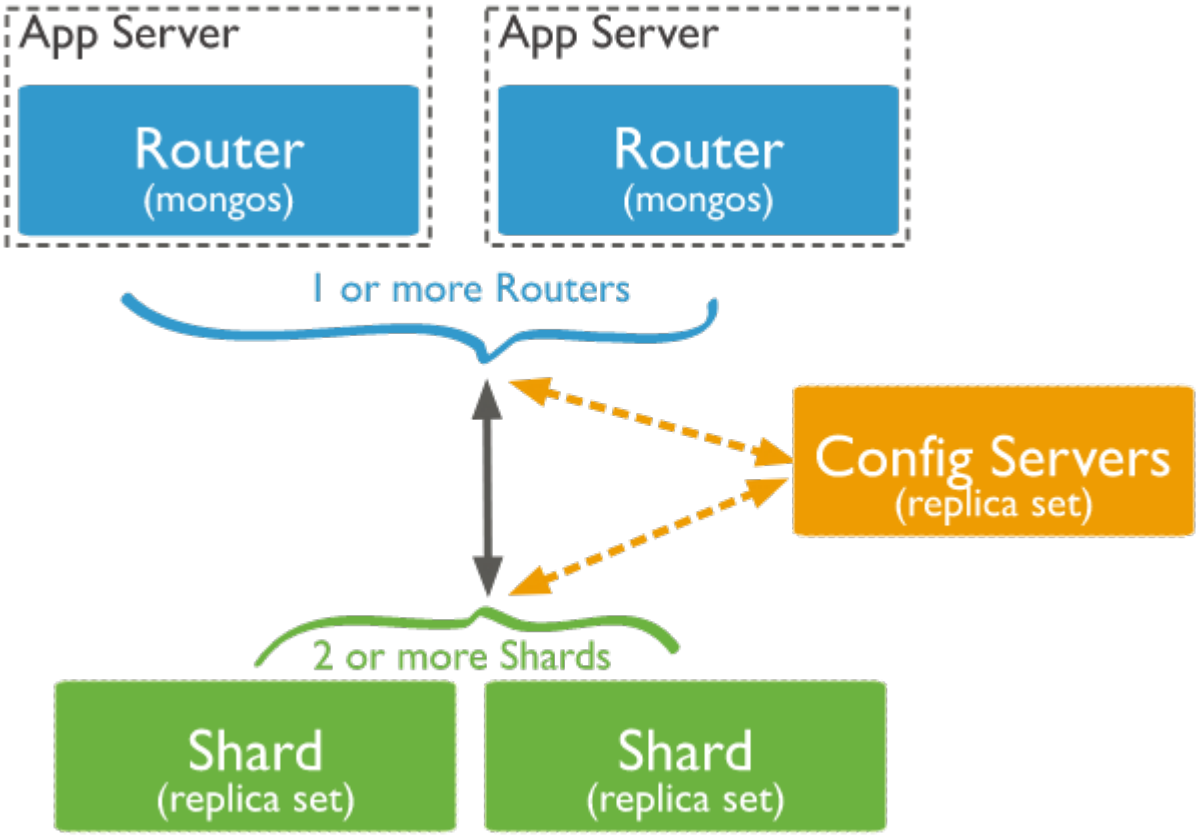
- Primary/Secondary Replication
- Sharding (horizontal scaling)

Indexing

- Can be used on one or several fields (implemented using B-trees)
- Also 2 dimensional spatial indexes for geometry-based data

Read more: https://docs.mongodb.com/manual/replication/ - https://docs.mongodb.com/manual/core/transactions/

MongoDB: Shard Cluster



Read more: https://docs.mongodb.com/manual/core/sharded-cluster-components/

Hands-on MongoDB
(Docker image)

MongoDB with Dockers

- We use the official container mongo preconfigured

```
$ docker pull mongo
```

- create a small network named `mongonet` with one server and one client

```
$ docker network create mongonet


$ docker run -it -p 27017:27017 --name mongo_server
      --network=mongonet mongo:latest
      /usr/bin/mongod --smallfiles --bind_ip_all


$ docker run -it --name mongo_cli
      --network=mongonet mongo:latest /bin/bash
```

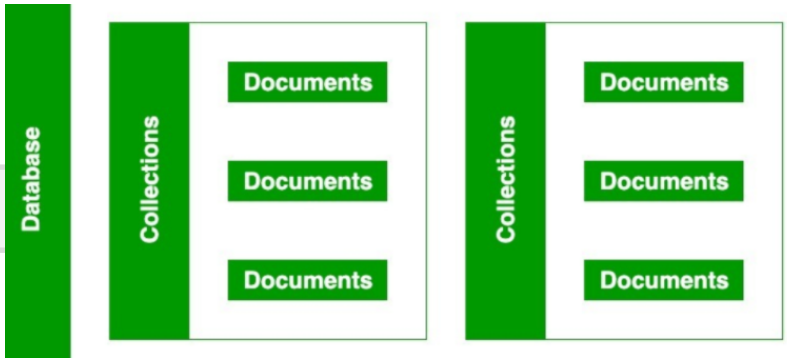
Mongo CLI: basic operations

- Use the command line interface on the client to connect to the mongo server

```
$ mongo mongo_server:27017
```

Create and switch to a new database

```
> use [databasename]
```



Insert a document: insert a document into a collection (e.g., named mycoll). The operation will create the collection if it does not exist yet.

```
> db.mycoll.insert(...)
```

Mongo CLI: Basic operations

Find documents: the find() method issues a query to retrieve data from a collection. All queries have the scope of a single collection.

- Queries can return all documents or only those matching a specific filter or criteria
- The find() method returns results in a cursor (an iterable object that yields documents)

```
> db.mycoll.find()


# filter the documents using the query operators {...}
> db.mycoll.find({ ... })
```

Mongo CLI: Query operators

```
# Exact match
> db.mycoll.find({"price" : 300 })
# Comparison (eq, gt, gte, lt, lte, in, nin):
> db.mycoll.find({"price" : { $gt: 300 } })
> db.mycoll.find({"year" : { $in: [2012, 2016] } })
# Existence (if document contains a field):
> db.mycoll.find({"discount" : { $exists: true } })
# logical (and, or, not, nor):
# AND:
> db.mycoll.find({field1 : {...}, field2 : {...} })
# OR:
> db.mycoll.find({
    $or: [{...}, {...}]
  })
```

https://docs.mongodb.com/manual/reference/operator/query/

Case Study (1)

We consider as use case a content management system, which needs to manage posts, images, videos, and so on, each with its own specific attributes.

```
> use cms
> db.cms.insert({ name : "hello world",
                  type : "post", size : 250,
                  comments : ["c1", "c2"] } )
> db.cms.insert({ name : "sunny day",
                  type : "image",
                  size : 300, url : "abc" })
> db.cms.insert({ name : "tutorial",
             type : "video", length : 125,
            path :"/video.flv",
            metadata : {quality : "480p",
            color : "b/n", private : false } })
```

Case Study (2)

```
> db.cms.find()

{ "_id" : ObjectId("5c76cafc8c3eb2953b2c8c2c"), "name" :
"hello world", "type" : "post", "size" : 250,
"comments" : [ "c1", "c2" ] }

{ "_id" : ObjectId("5c76cb0a8c3eb2953b2c8c2d"), "name" :
"sunny day", "type" : "image",
"size" : 300, "url" : "abc" }

{ "_id" : ObjectId("5c76cb158c3eb2953b2c8c2e"), "name" :
"tutorial", "type" : "video", "length" : 125, "path" : "/
video.flv", "metadata" : { "quality" : "480p", "color" : "b/
n", "private" : false } }
```

Case Study (3)

```
> db.cms.find( {size : { $gt : 100 } } )


{ "_id" : ObjectId("5c76cafc8c3eb2953b2c8c2c"), "name" : "hello
world", "type" : "post", "size" : 250,
"comments" : [ "c1", "c2" ] }
{ "_id" : ObjectId("5c76cb0a8c3eb2953b2c8c2d"),
"name" : "sunny day", "type" : "image", "size" : 300,
"url" : "abc" }


> db.cms.find( {size : { $lt : 100 } } )
> db.cms.find({ length : { $exists: true } } )


{ "_id" : ObjectId("5c76cb158c3eb2953b2c8c2e"),
"name" : "tutorial", "type" : "video", "length" : 125, "path" : "/
video.flv", "metadata" : { "quality" : "480p", "color" : "b/n",
"private" : false } }
```

Case Study (4)

```
> db.cms.findOne({comments:{ $exists: true }})
{       "_id" : ObjectId("5c76cafc8c3eb2953b2c8c2c"),
        "name" : "hello world",
        "type" : "post",
        "size" : 250,
        "comments" : [
                "c1",
                "c2"
        ]
}
> db.cms.findOne({ comments:
                { $exists: true}}).comments[1]
  c2
```

Case Study (5)

```
> db.cms.find({ comments : { $exists: true }})

{ "_id" : ObjectId("5c76cafc8c3eb2953b2c8c2c"), "name" : "hello
world", "type" : "post", "size" : 250,
"comments" : [ "c1", "c2" ] }

> db.cms.find({ "comments.2": { $exists: true }})

> db.cms.find({ "comments.1": { $exists: true }})

{ "_id" : ObjectId("5c76cafc8c3eb2953b2c8c2c"), "name" : "hello
world", "type" : "post", "size" : 250,
"comments" : [ "c1", "c2" ] }
```

Case Study (6)

```
> db.cms.find( {size:{ $gt : 100 },
                                    type: "image"})
{ "_id" : ObjectId("5c76cb0a8c3eb2953b2c8c2d"), "name" :
"sunny day", "type" : "image", "size" : 300, "url" : "abc" }


> db.cms.find( { $or : [{size : { $gt : 100 }},
                                                {type :
"image"}]})


{ "_id" : ObjectId("5c76cafc8c3eb2953b2c8c2c"), "name" :
"hello world", "type" : "post", "size" : 250, "comments" : [
"c1", "c2" ] }
{ "_id" : ObjectId("5c76cb0a8c3eb2953b2c8c2d"), "name" :
"sunny day", "type" : "image", "size" : 300, "url" : "abc" }
```

Mongo CLI: Query operators

Sort query results: to specify an order for the result set, append the sort() method to the query.

- Pass to sort() a document which contains the field(s) to sort by and the corresponding sort type (1 for ascending, -1 for descending)

```
> db.mycoll.find().sort( { "name" : 1 } )
```

https://docs.mongodb.com/manual/reference/operator/query/

Case Study (7)

```
> db.cms.find().sort({ name : 1 })


{ "_id" : ObjectId("5c76cafc8c3eb2953b2c8c2c"),
"name" : "hello world", "type" : "post", "size" : 250,
"comments" : [ "c1", "c2" ] }


{ "_id" : ObjectId("5c76cb0a8c3eb2953b2c8c2d"),
"name" : "sunny day", "type" : "image", "size" : 300, "url" :
"abc" }


{ "_id" : ObjectId("5c76cb158c3eb2953b2c8c2e"),
"name" : "tutorial", "type" : "video",
"length" : 125, "path" : "/video.flv",
"metadata" : { "quality" : "480p", "color" : "b/n",
"private" : false } }
```

Mongo CLI: Basic operations

Update a document: using update(); several update operators are available in mongo.

$set sets the value of a field in a document. The update can be applied to one or multiple occurrencies that matches the update filter.

```
> db.mycoll.update(
      { field : value },
      { $set:
            { "address.street": "East 31st Street" }
} )
```

update filter

Update multiple occurrences

```
> db.mycoll.update(
      { field : value },
      { $set: { ... } },
      {multi: true} )
```

https://docs.mongodb.com/manual/reference/operator/update/

Case Study (8)

```
> db.cms.update({ "name" : "Canon EOS 750D" }, {$set: {"address.street":"East 31st Street"}})
WriteResult({"nMatched":0, "nUpserted":0, "nModified":0})

> db.cms.update({"name" : "mycms-logo"},
                          {$set: {"metadata.quality": "hd"}})
WriteResult({ "nMatched":0, "nUpserted":0, "nModified":0})

> db.cms.find({name : "mycms-logo"})

> db.cms.find({type : "image"})

{ "_id" : ObjectId("5c76cb0a8c3eb2953b2c8c2d"), "name" : "sunny day", "type" : "image", "size" : 300, "url" : "abc" }
```

Case Study (9)

```
> db.cms.update({type : "image"},  {$set:
{"metadata.author": "myname"}} , {multi: true})


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


>  db.cms.find({type : "image"})


{ "_id" : ObjectId("5c76cb0a8c3eb2953b2c8c2d"), "name" :
"sunny day", "type" : "image", "size" : 300, "url" : "abc",
"metadata" : { "author" : "myname" } }


> db.cms.find({ type : "post" })


{ "_id" : ObjectId("5c76cafc8c3eb2953b2c8c2c"), "name" :
"hello world", "type" : "post", "size" : 250, "comments" : [
"c1", "c2" ] }
```

Case Study (10)

```
> db.cms.update({ name : "hello world"},
      {$push: {"comments": "a new comment"}})

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

> db.cms.update({ name : "hello world" },
{$push: {"comments": "a second new comment"}})

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

> db.cms.find({ type : "post"})
{ "_id" : ObjectId("5c76cafc8c3eb2953b2c8c2c"), "name" :
"hello world", "type" : "post", "size" : 250, "comments" : [
"c1", "c2", "a new comment", "a second new comment" ] }
```

Mongo CLI: Basic operations

Remove documents: the remove() method removes documents from a collection. The method takes a conditions document that determines the documents to remove

```
> db.mycoll.remove(
      { "borough": "Manhattan" } )
> db.mycoll.remove(
      { "borough": "Queens" },
      { justOne: true } )


# remove all documents:
> db.mycoll.remove( { } )
```

https://docs.mongodb.com/manual/reference/operator/update/

Mongo CLI: Basic operations

Drop a collection: to remove all documents from a collection (and the collection itself), the drop() operation should be used.

```
> db.mycoll.drop()
```

https://docs.mongodb.com/manual/reference/operator/update/

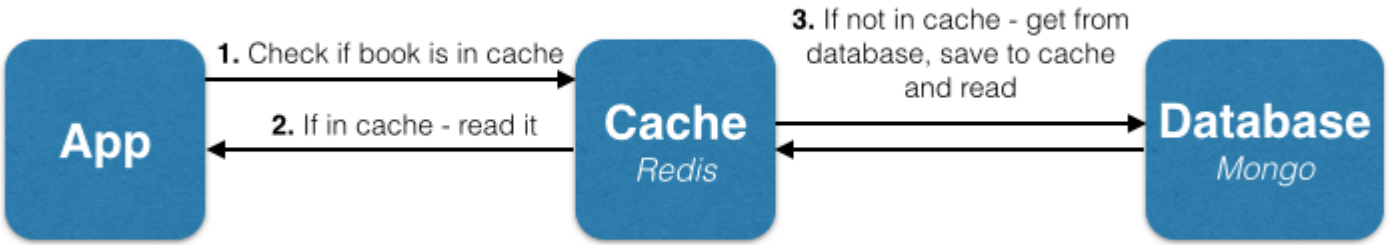
Different needs, different solutions

- When storing data, it is best to use multiple data storage technologies
  - Chosen upon the way data is being used

A simple yet effective use case:

- A simple web library, which interacts with a (persistent) database
- the communication with the database can cause a big overhead

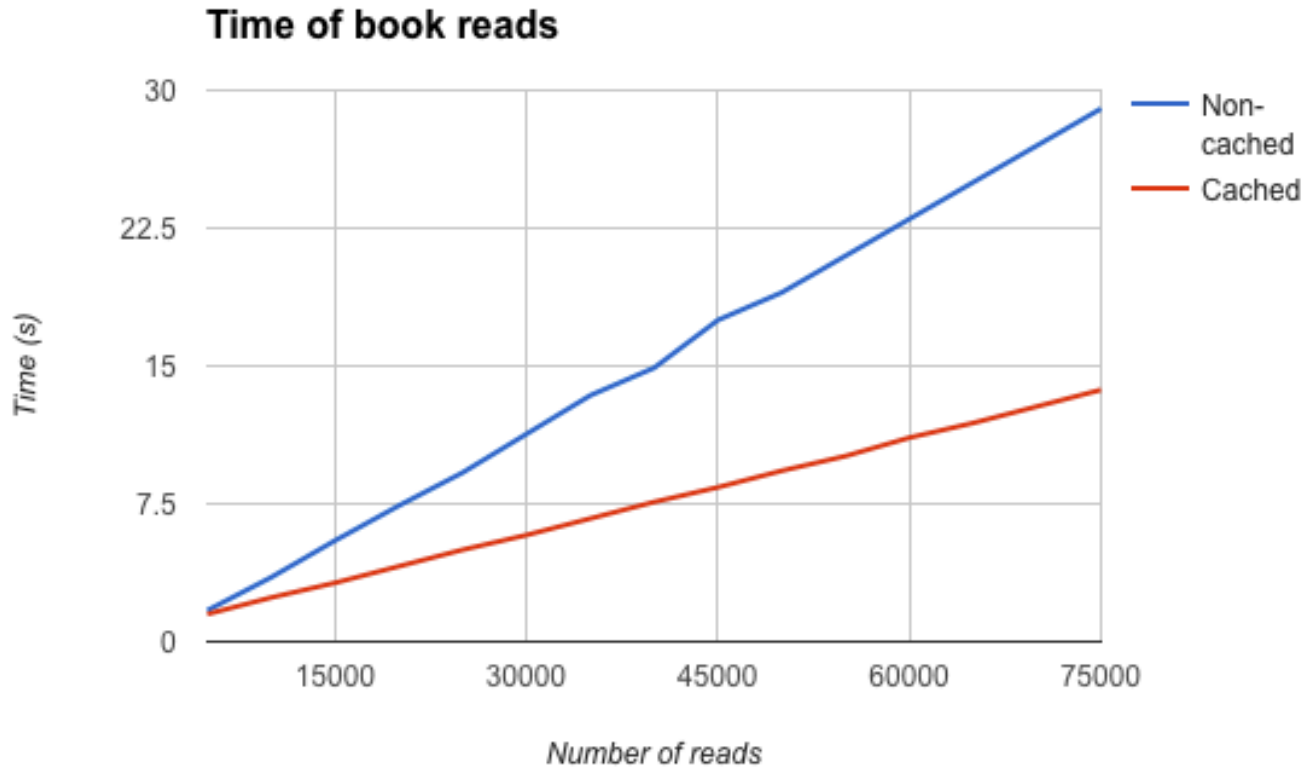
Solutions?
Use an in-memory key-value store as **caching** system!



Read more: https://www.sitepoint.com/caching-a-mongodb-database-with-redis/

Different needs, different solutions

- Case study: the management of a library
- Books are stored in a Mongo database
- A web application can access and read books



Read more: https://www.sitepoint.com/caching-a-mongodb-database-with-redis/