Università degli Studi di Roma “Tor Vergata”
Dipartimento di Ingegneria Civile e Ingegneria Informatica

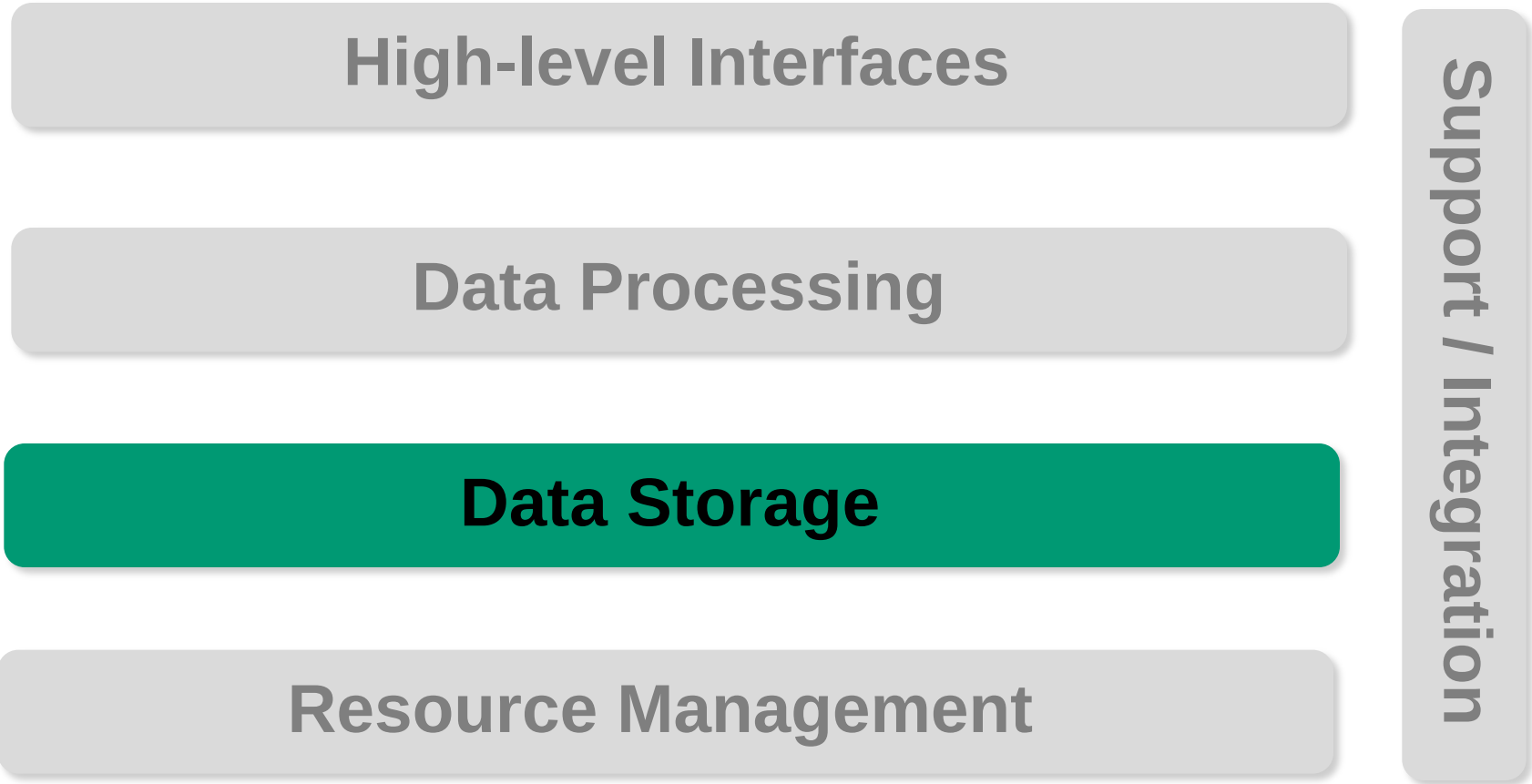
NoSQL: Redis and MongoDB

A.A. 2016/17

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

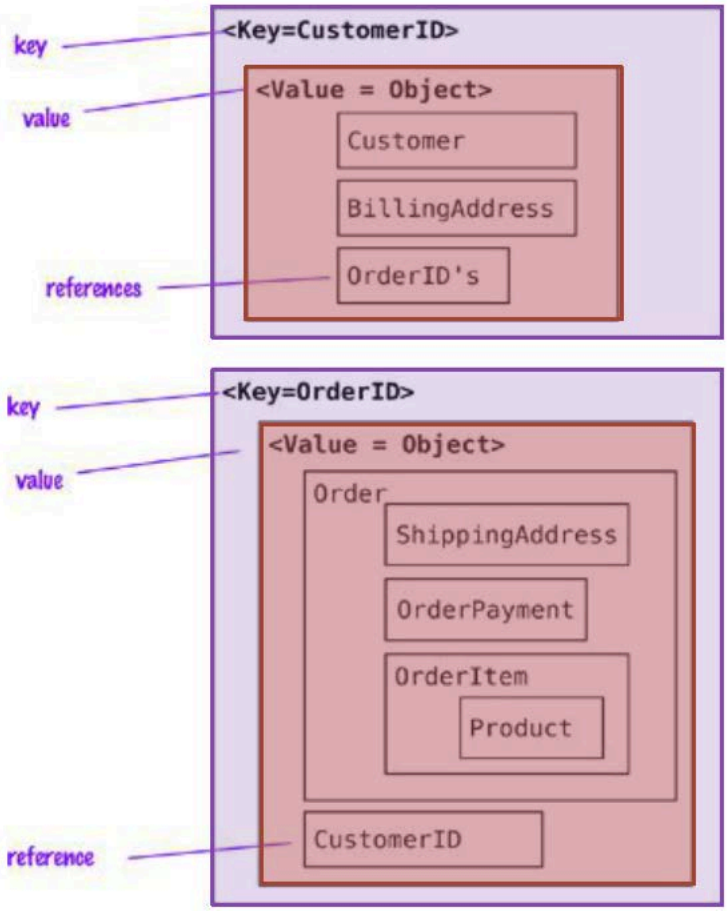
Data models for NoSQL systems:

- 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



Suitable use cases for key-value data stores

- Storing session information in web apps
  - Every session is unique and is assigned a unique sessionId value
  - Store everything about the session using a single put, request or retrieved using get
- User profiles and preferences
  - Almost every user has a unique userId, username, …, as well as preferences such as language, which products the user has access to, …
  - Put all into an object, so getting preferences of a user takes a single get operation
- Shopping cart data
  - All the shopping information can be put into the value where the key is the userId

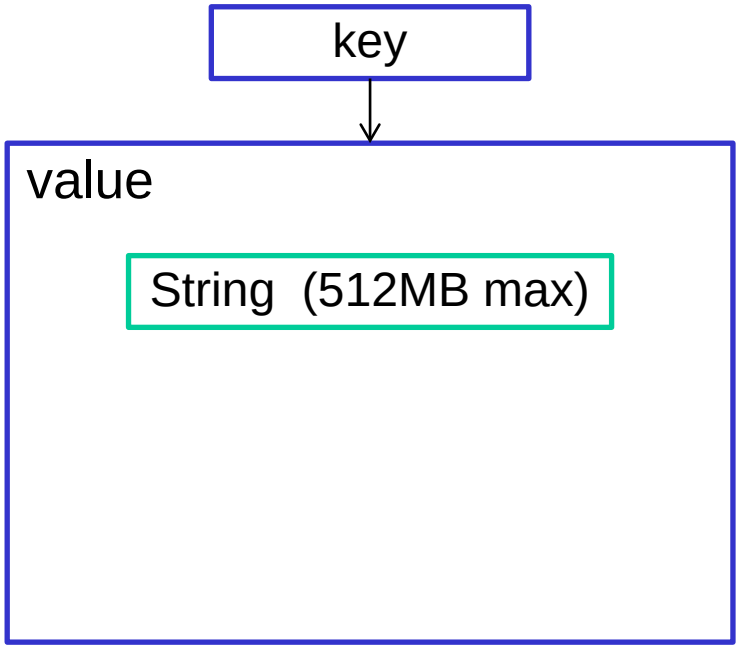
Redis

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

- Redis was the most popular implementation of a key-value database as of August 2015, according to DB-Engines Ranking.

Data Model

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



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

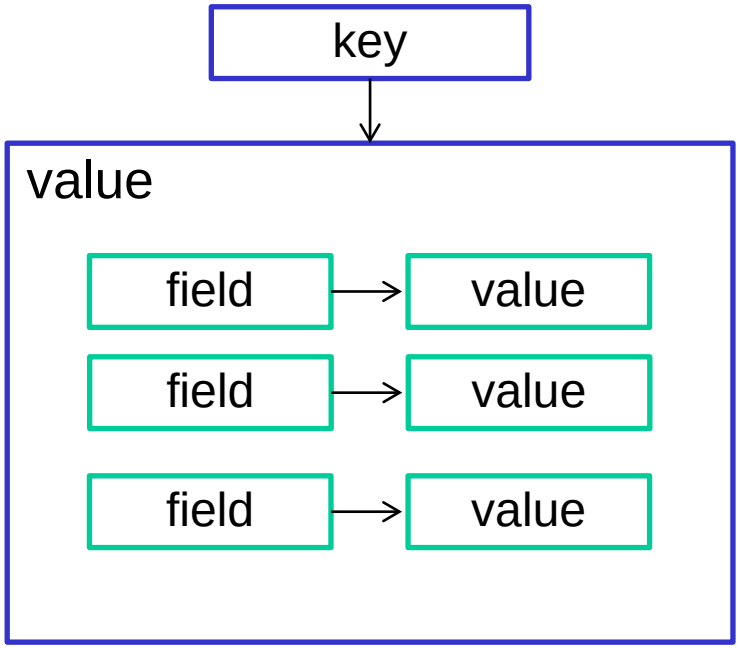
Redis

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

- Redis was the most popular implementation of a key-value database as of August 2015, according to DB-Engines Ranking.

Data Model

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



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

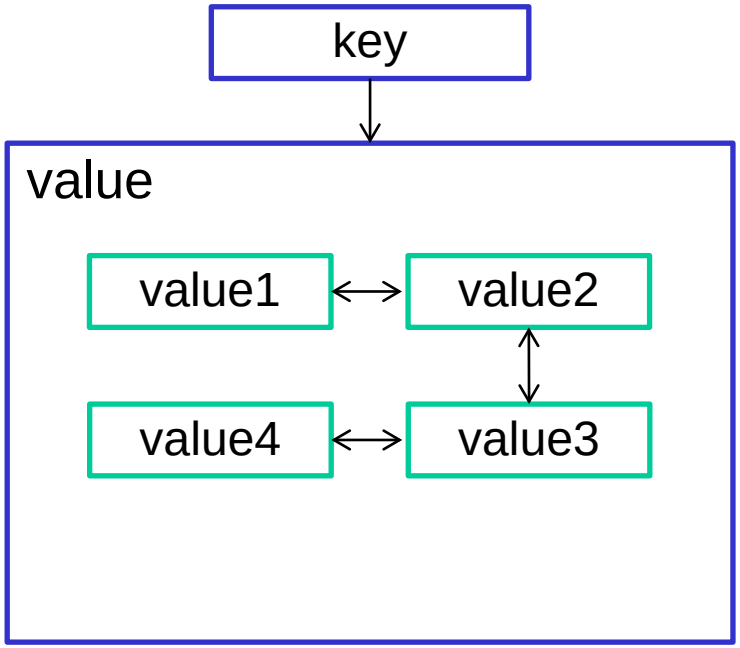
Redis

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

- Redis was the most popular implementation of a key-value database as of August 2015, according to DB-Engines Ranking.

Data Model

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



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

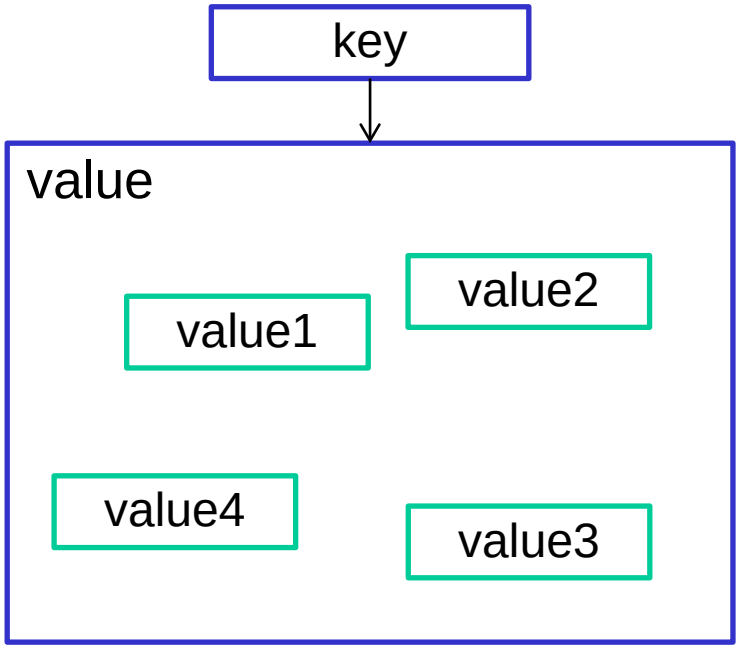
Redis

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

- Redis was the most popular implementation of a key-value database as of August 2015, according to DB-Engines Ranking.

Data Model

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



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

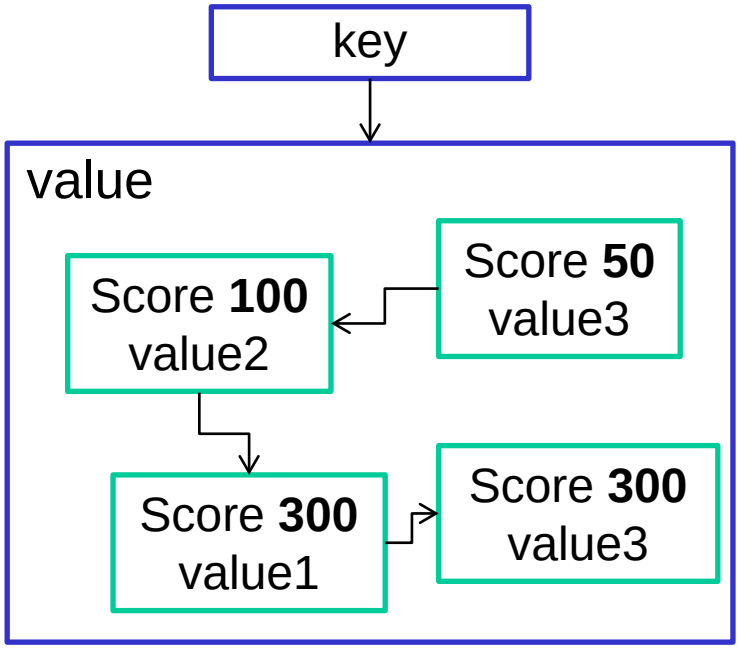
Redis

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

- Redis was the most popular implementation of a key-value database as of August 2015, according to DB-Engines Ranking.

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/

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/

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

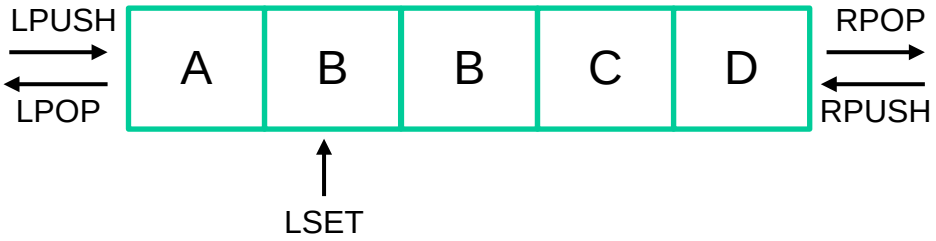
The presence of a score enables to rank or to retrieve the elements as well as changing their order during the lifetime of the sorted set

```
# 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/

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-besed indexes)
redis> LRANGE key start stop
```

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

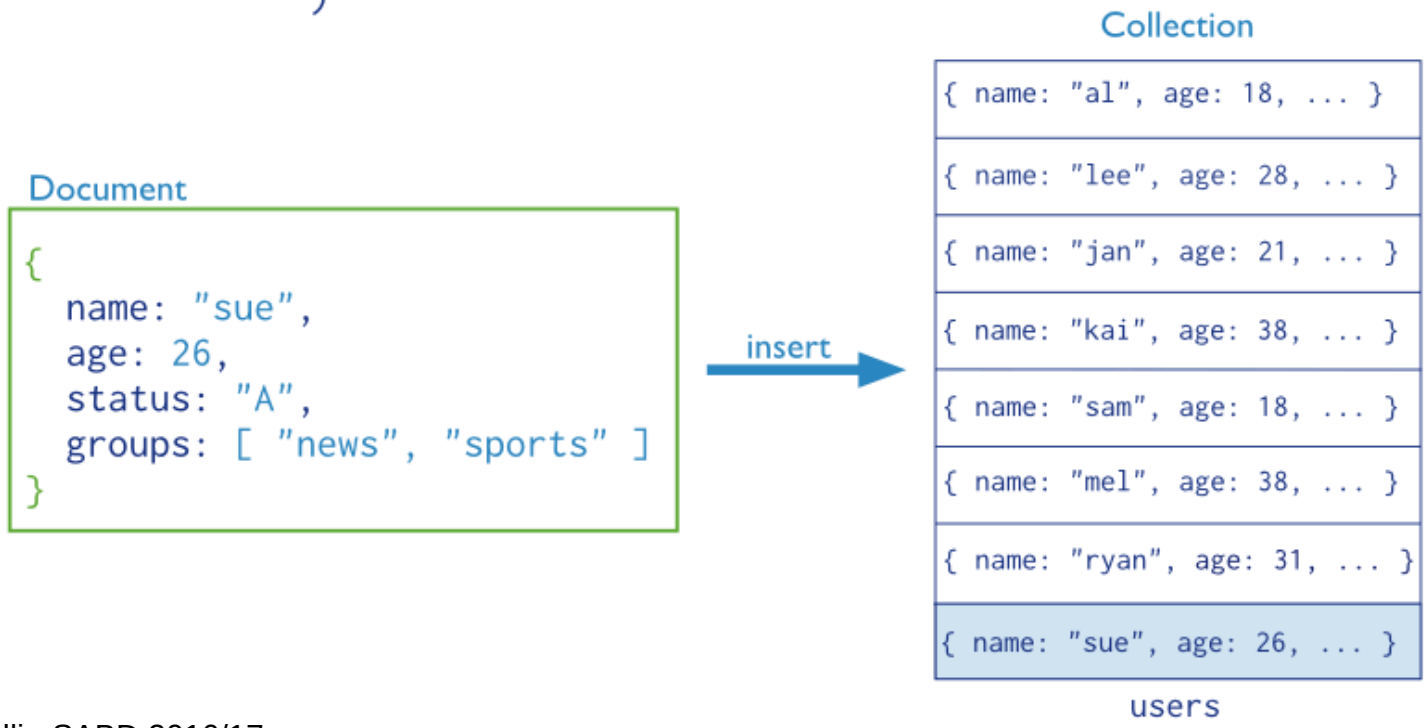
Suitable use cases for document data stores

- Applications dealing with data that can be easily interpreted as documents
  - A blog post or an item in a CMS
  - Contents (images, posts) can be transformed into a document format, even though they have different attributes
- Catalogs
  - Flexible schema makes it particularly well suited to store information of products
  - Easy to store and query a set of attributes for entities such as people, places, and products
- Customized user experience
- Model and store machine generated data
  - log events or monitor information
  - events from different sources carry different information

These pieces of information are mainly manipulated as aggregates and do not have many relationships with other data.

In MongoDB:

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



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

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


$ 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/

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/

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/

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/

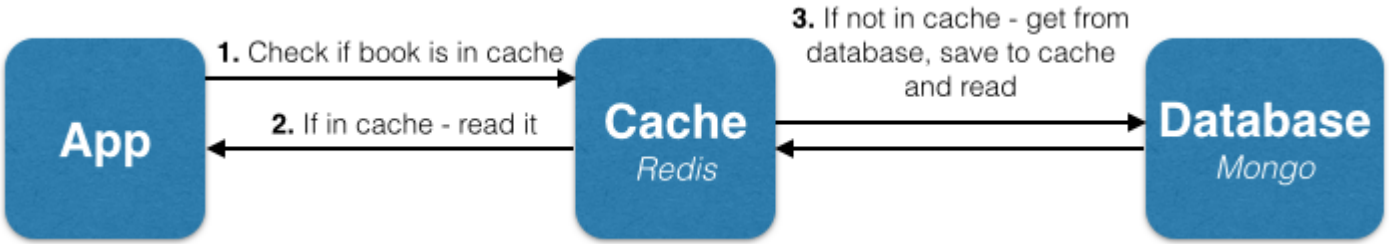
Different needs, different solutions

- 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

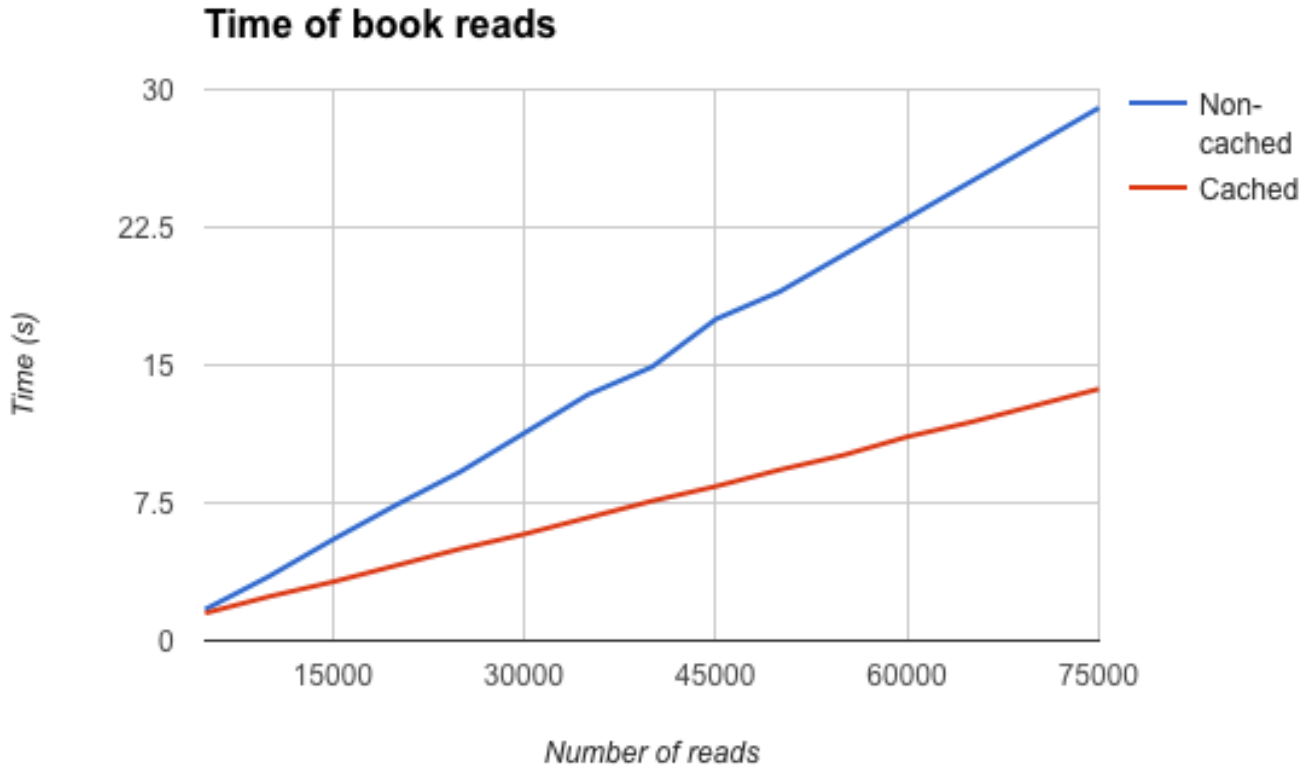
Solutions?
Use an in-memory key-value store as **caching** system!



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/