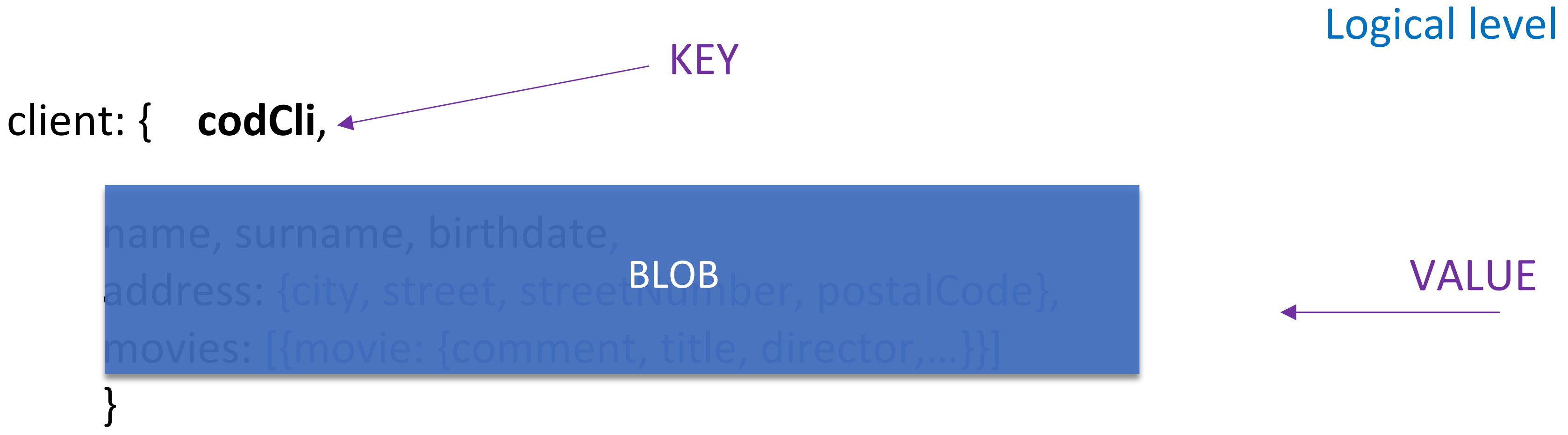
Key-value NoSQL data stores

Key-value data model at a glance

- Each data instance is represented in the form (key, value)
  - key is an identifier
  - value is the aggregate
- The aggregate is totally opaque at the logical level
  - just a big blob of mostly meaningless bit, that the system just stores
  - values do not have a known structure
  - very high flexibility: arbitrary aggregate content
- The aggregate is visible at the application level
- Data instances can be grouped into logical collections



Logical level

KEY

client: { **codCli**,

name, surname, birthdate,
address: {city, street, streetNumber, postalCode},
movies: [{movie: {comment, title, director,...}}]
}

BLOB

VALUE

Key-value data model at a glance

- Each data instance is represented in the form (key, value)
  - key is an identifier
  - value is the aggregate
- The aggregate is totally opaque at the logical level
  - just a big blob of mostly meaningless bit, that the system just stores
  - values do not have a known structure
  - very high flexibility: arbitrary aggregate content
- The aggregate is visible at the application level
- Data instances can be grouped into logical collections

Logical level

KEY

client: { **codCli**,

name, surname, birthdate,
address: {city, street, streetNumber, postalCode},
movies: [{movie: {comment, title, director,…}}]
}

VALUE

Instance structure

- No schema information
- No nested values are visible at the NoSQL system level but only at the application level

Collections

- Pairs can be grouped into logical collections/namespaces: sets of aggregates, i.e., sets of (key, value) pairs

key	value
key	value
key	value
key	value

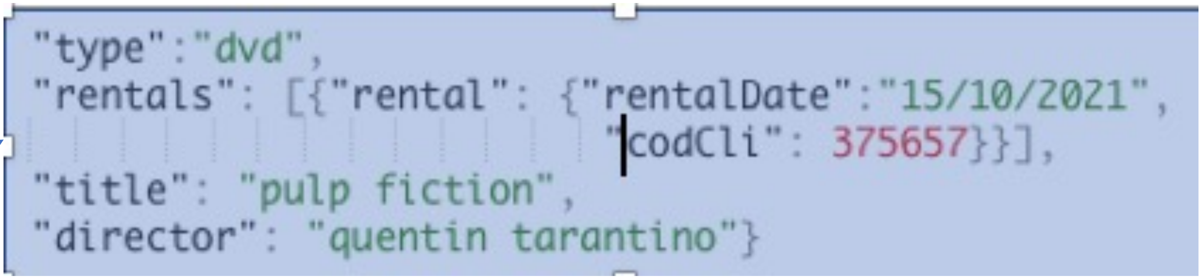
In RDBMS world: A table with two columns:

ID column (primary key)

DATA column storing the value (unstructured BLOB)

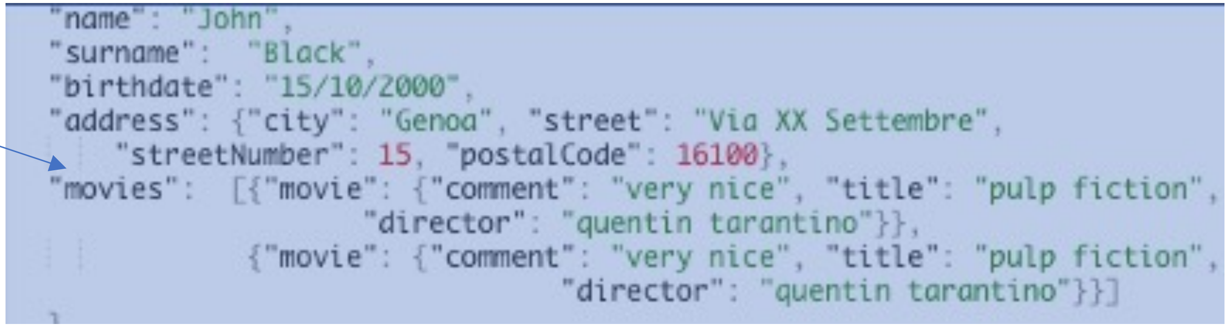
A

1	V1
2	V2
3	V3
4	V4



```
"type":"dvd",
"rentals": [{"rental": {"rentalDate":"15/10/2021",
                        "codCli": 375657}}],
"title": "pulp fiction",
"director": "quentin tarantino"}
```

…



```
"name": "John",
"surname":  "Black",
"birthdate": "15/10/2000",
"address": {"city": "Genoa", "street": "Via XX Settembre",
    "streetNumber": 15, "postalCode": 16100},
"movies":  [{"movie": {"comment": "very nice", "title": "pulp fiction",
                  "director": "quentin tarantino"}},
              {"movie": {"comment": "very nice", "title": "pulp fiction",
                              "director": "quentin tarantino"}}]
```

(see buckets in Riak)

Collections and schema information

- The only «schema» information the system is aware of is the collection
- There is no structure specification nor schema constraint in the data model
- But at application-level, we can take advantage of some regularities in data, if any

Collections

- Pairs can be grouped into logical collections/namespaces: sets of aggregates, i.e., sets of (key, value) pairs

key	value
key	value
key	value
key	value

Videos

1234	value
5678	value
9999	value
0000	value

Clients

375657	value
375658	value
375659	value
375660	value

(see buckets in Riak)

Keys

- (key, value) pair
- At the logical level: identification + data retrieval
  - the key needs to assume unique values in the collection (i.e., it is an identifier)
- At the physical level, the key tells the system how to partition the data and where to store the data
  - partition key
- The partition key and the identifier/primary key coincide
- Aggregates can be directly retrieved only by specifying values for attributes in the partition key
- *A simple hash table (map), primarily used when all accesses to the data are via the key*

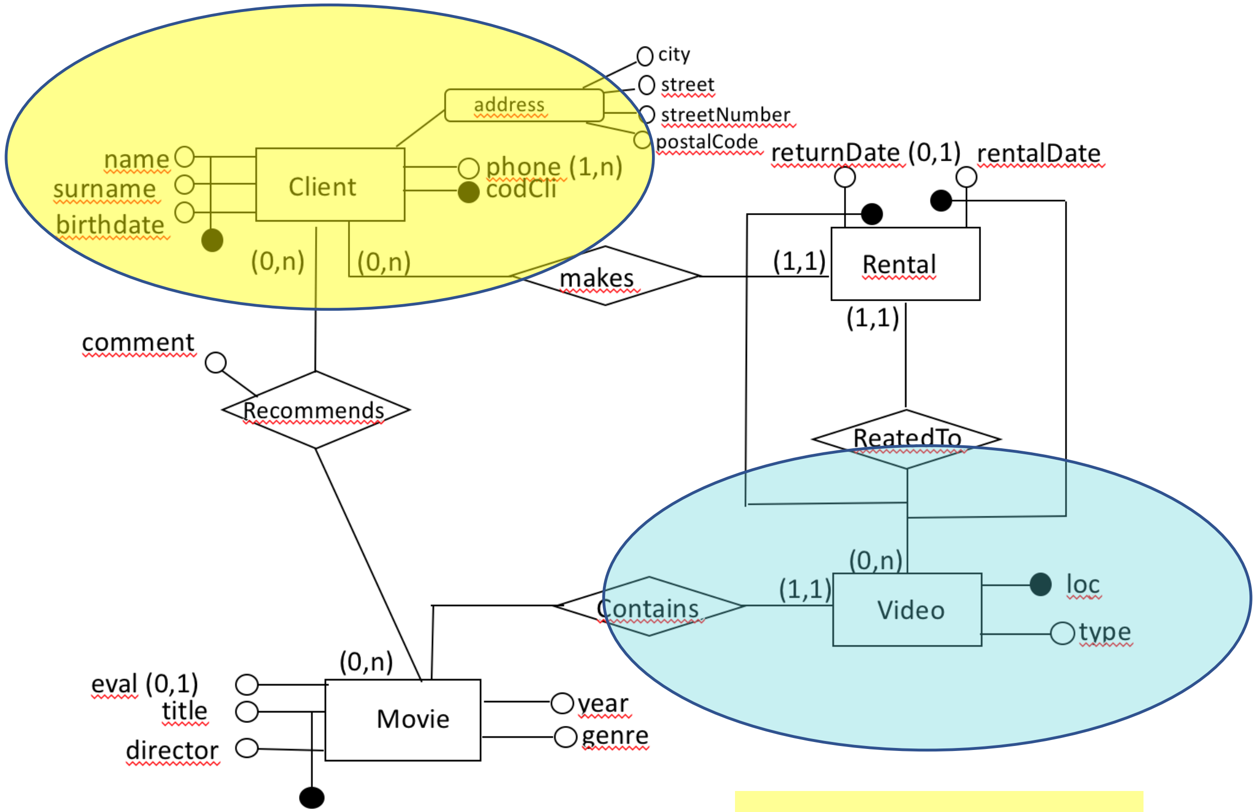
Keys

- How to design the key?
  - Provided by the user (natural unique key): userID, e-mail,...
  - Generated by some algorithm
  - Derived from time-stamps (or other data)

- Use cases with natural keys:
  - user profiles (user ID), …
  - shopping cart data (user ID)
  - web session data (with the session ID as the key)

- Expiration of keys
  - After a certain time interval
    - e.g. for caches, session/shopping cart objects,...

Keys



Video: {loc: …}

Client: {codCli: …}

Videos

1234	value
5678	value
9999	value
0000	value

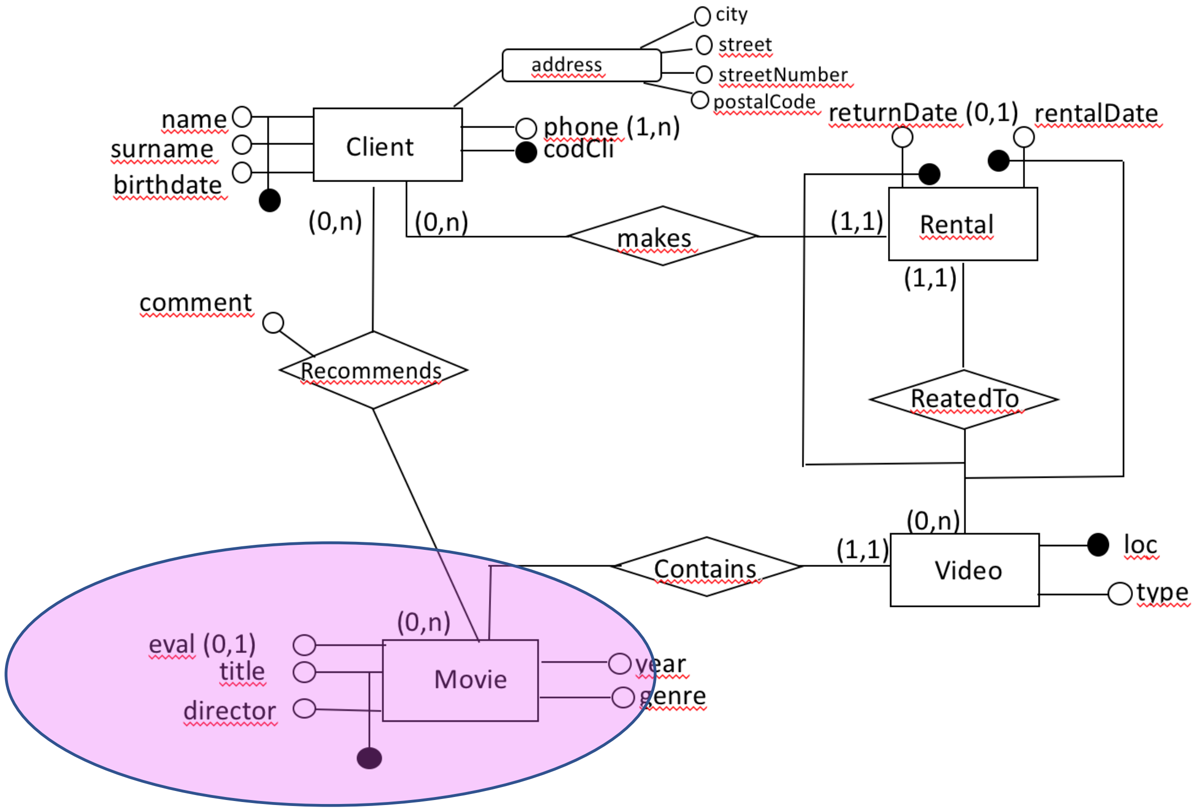
Clients

375657	value
375658	value
375659	value
375660	value

Key is loc

Key is codeCli

Keys



There is no single unique attribute, different options are possible

1. Id
2. (title and director are included in the value)

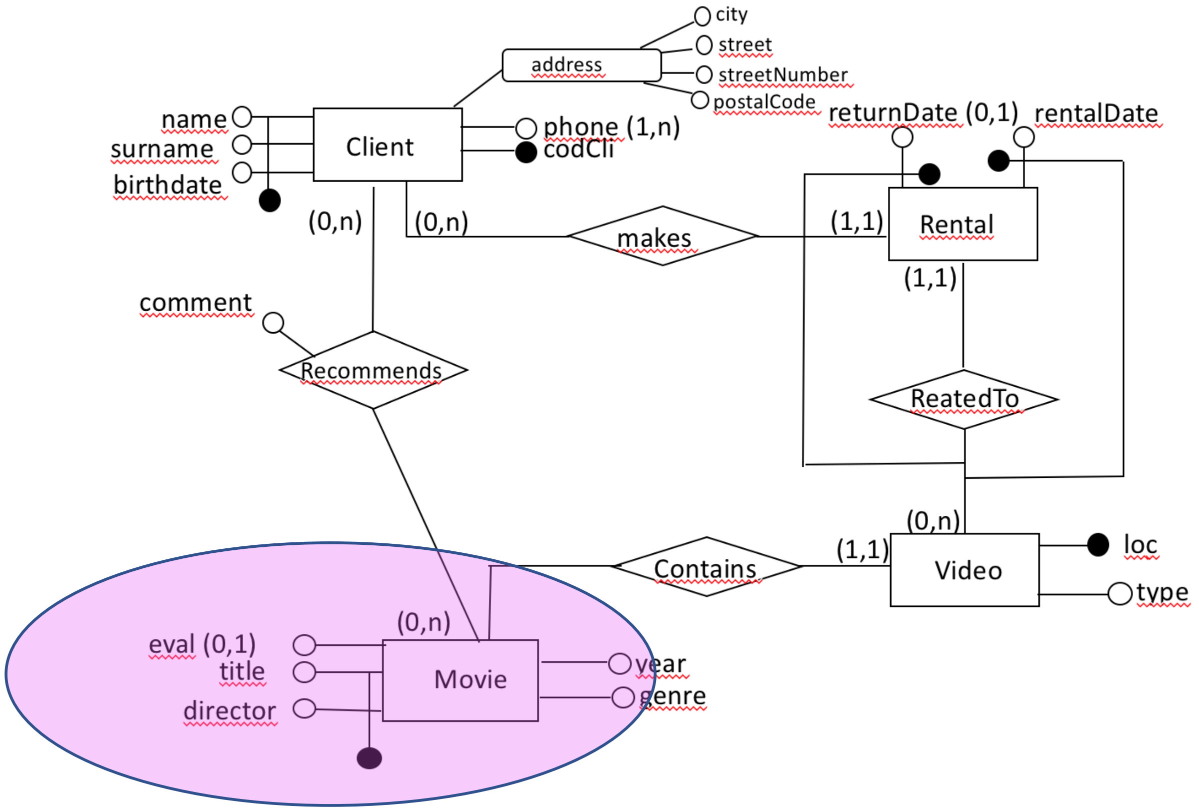
Movie : {id: …}

Movies

1	value
2	value
3	value
4	value

Key is id

Keys



There is no single unique attribute, different options are possible

2. title:director

Movie : {'title:director': …}

Movies

pulp fiction:quentin tarantino	value
kill bill:quentin tarantino	value
le iene:quentin tarantino	value
dumbo : gabriele salvatores	value

Key is title:director

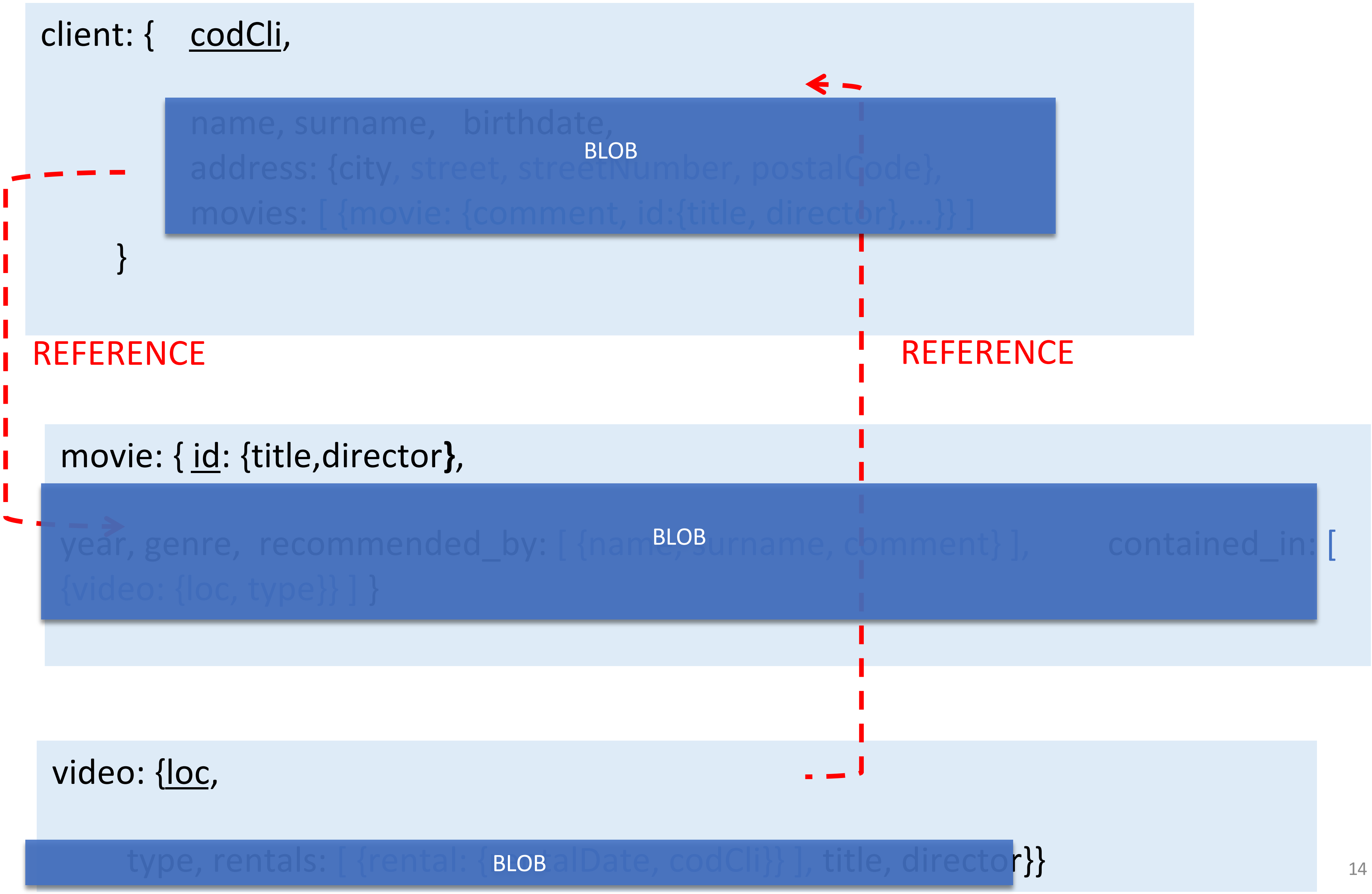
Identifier vs partition/sharding key

- In the Videos collection, videos are partitioned by loc
- In the Clients collection, clients are partitioned by codCli
- In the Movies collection, clients are partitioned by
  - Id
  - Title:director

- This impacts the way data are stored
- This impacts the way data can be retrieved

Video Rental Example

Logical level


client: { codCli,

name, surname, birthdate,

address: {city, street, streetNumber, postalCode},

movies: [{movie: {comment, id:{title, director},...}}]

}

BLOB

REFERENCE

REFERENCE

movie: { id: {title,director},

year, genre, recommended_by: [{name, surname, comment}], contained_in: [{video: {loc, type}}] }

BLOB

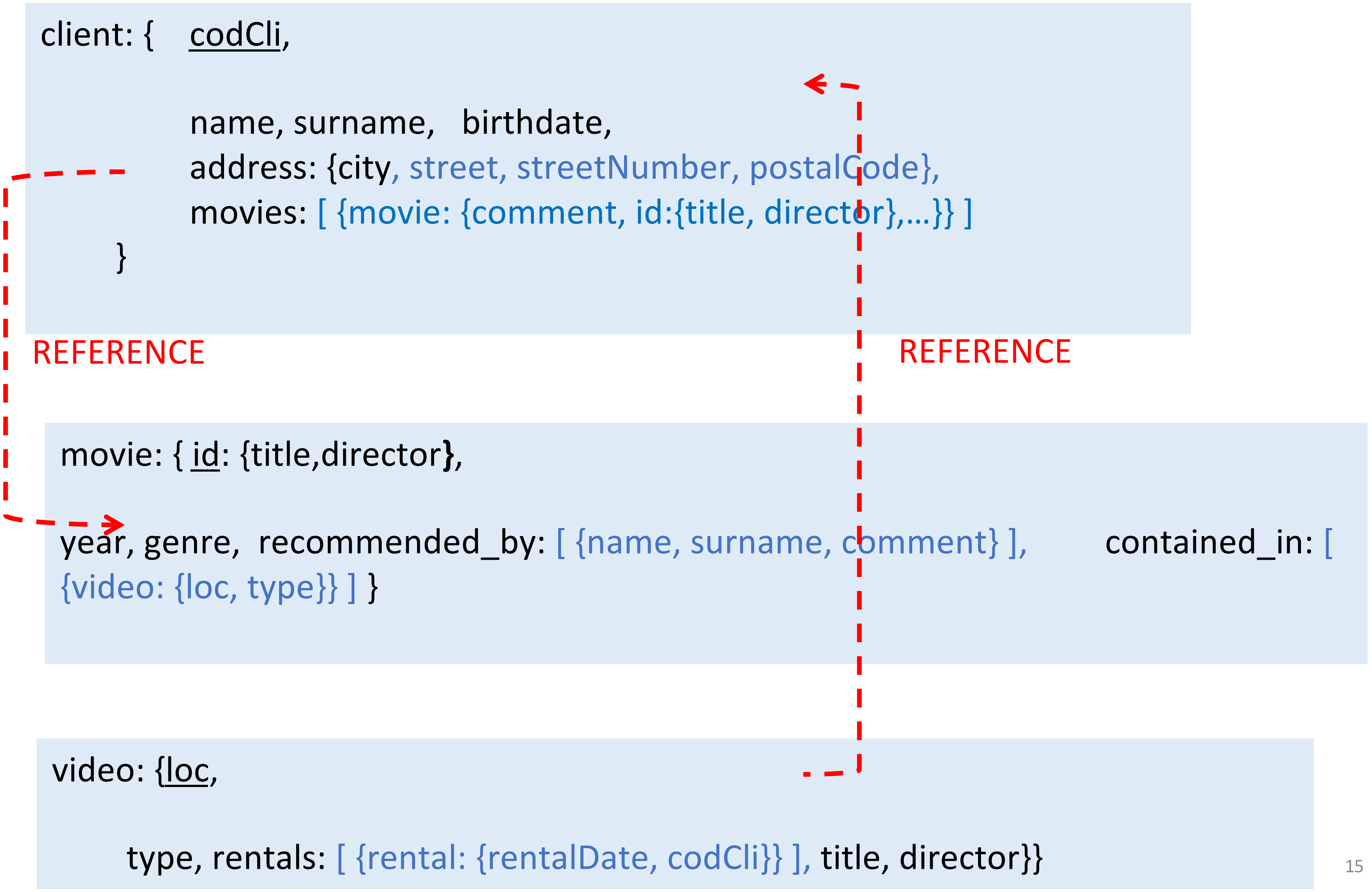
video: {loc,

type, rentals: [{rental: {rentalDate, codCli}}], title, director}}

BLOB

Video Rental Example

Application level



client: { codCli,

name, surname, birthdate,
address: {city, street, streetNumber, postalCode},
movies: [{movie: {comment, id:{title, director},…}}]
}

REFERENCE

REFERENCE

movie: { id: {title,director},

year, genre, recommended_by: [{name, surname, comment}], contained_in: [{video: {loc, type}}] }

video: {loc,

type, rentals: [{rental: {rentalDate, codCli}}], title, director}}

Interaction

- Basic operations:
  - Put a value for a key (if the key already exists the corresponding value is overwritten)

    `put(key, value)`
  - Get the value for the key

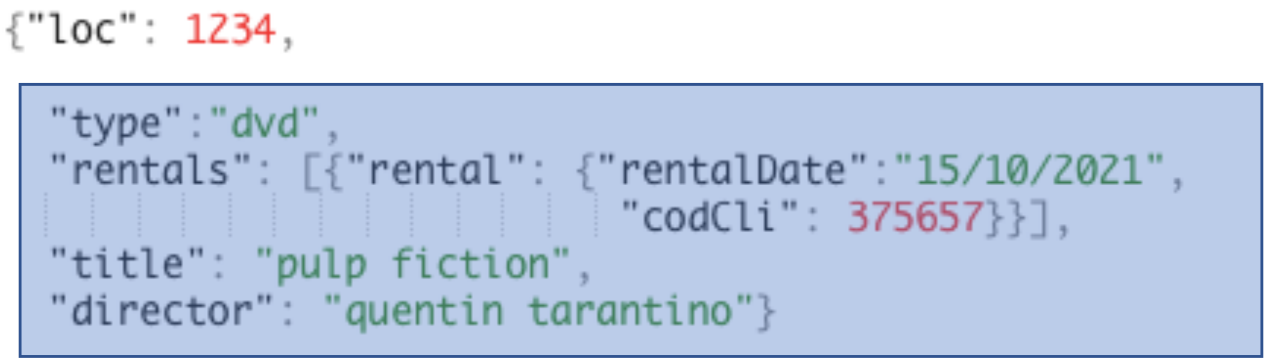
    `value:= get(key)`
  - Delete a key (and the corresponding value)

    `delete(key)`

Interaction

- Lookup based on the key
- The application can read the entire aggregate by using the key
- Queries with respect to specific aggregate fields are not supported (in general): we need to read the whole aggregate and check query conditions at the application level
- Associations navigated by sequence of lookups
- In case collections are supported, key values are preceded by the collection name in each operation

Example 1 – Find the title of the movie in a video

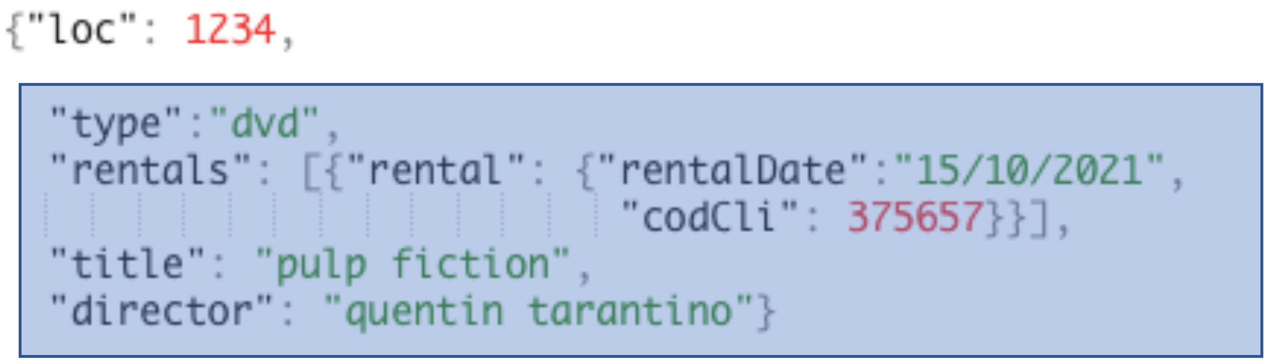


```
{"loc": 1234,
 "type":"dvd",
 "rentals": [{"rental": {"rentalDate":"15/10/2021",
                         "codCli": 375657}}],
 "title": "pulp fiction",
 "director": "quentin tarantino"}
```

get(Videos,1234)

- at the data store level, find Video 1234 in namespace Videos, no detailed information about its content
- at the *application level*, we go inside the aggregate value and we discover that it contains «Pulp Fiction»

Example 2 – Find the videos containing a certain movie



```
{"loc": 1234,
  "type":"dvd",
  "rentals": [{"rental": {"rentalDate":"15/10/2021",
                          "codCli": 375657}}],
  "title": "pulp fiction",
  "director": "quentin tarantino"}
```

get(Videos,???)

- We cannot filter on other attributes than the key
- At the data store level, can only get all the videos, and
- At the application level, filter them by titles and director

Example 2 – Find the videos containing a certain movie - we should use the movies collection instead

get(Movies,pulp fiction:quentin tarantino)

- At the data store level, get the value
- At application level, find *contained_in* and get the location

Movies collection
Key: title:director,
Value contains {year, genre,
            recommended_by: [{name, surname, comment}],
            contained_in: [{video: {loc, type}}]}

REMARK: the relationship is part of the aggregate, a single data access to retrieve all the relevant information (=a single data node)

Example 2 – Find the videos containing a certain movie - we should use the movies collection instead, what if we chose the other key?

```
Movies collection
Key:MovieID,
Value contains {ttle, drector, year, genre,
          recommended_by: [{name, surname, comment}],
          contained_in: [{video: {loc, type}}]}
```

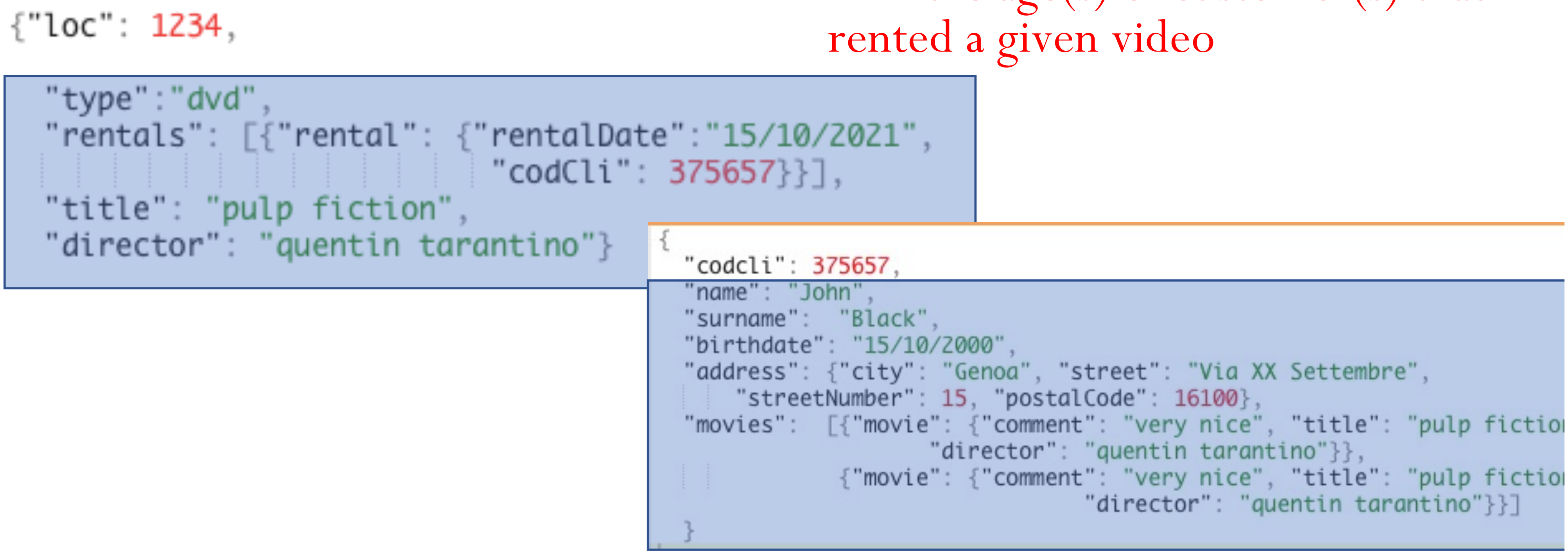
get(Movies,???)

- We cannot filter on other attributes than the key
- At the data store level, can only get all the movies, and
- At the application level, filter them by titles and director

REMARK: still likely more efficient than starting from Videos, since Movies are less than videos and all the videos containing a certain movie are stored inside the same aggregate

Example 3 - Relationships

Find the age(s) of customer(s) that rented a given video



```
{"loc": 1234,
  "type":"dvd",
  "rentals": [{"rental": {"rentalDate":"15/10/2021",
                          "codCli": 375657}}],
  "title": "pulp fiction",
  "director": "quentin tarantino"}
```

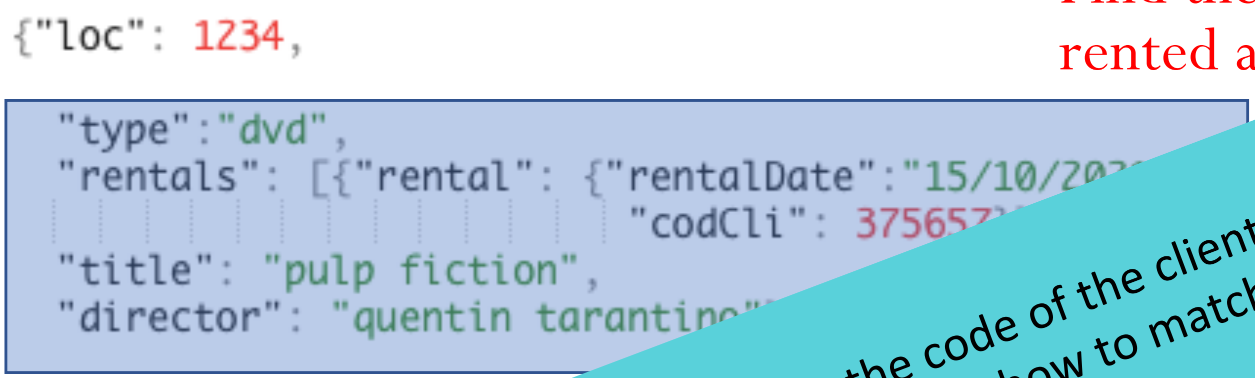
```
{
  "codcli": 375657,
  "name": "John",
  "surname":  "Black",
  "birthdate": "15/10/2000",
  "address": {"city": "Genoa", "street": "Via XX Settembre",
      "streetNumber": 15, "postalCode": 16100},
  "movies":  [{"movie": {"comment": "very nice", "title": "pulp fictio
                          "director": "quentin tarantino"}},
             {"movie": {"comment": "very nice", "title": "pulp fictio
                                  "director": "quentin tarantino"}}]
  }
```

get(Videos,1234)

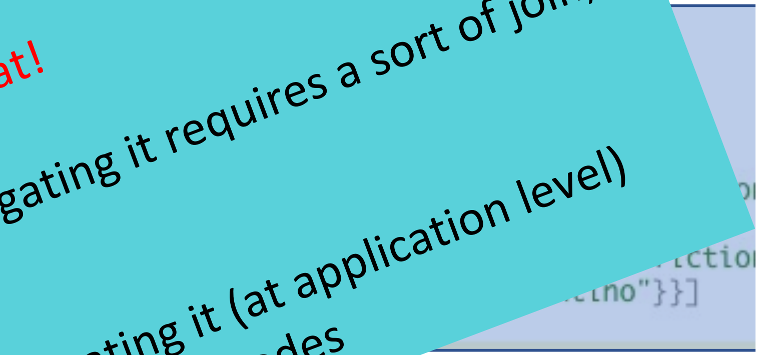
- At the data store level, find Video 1234 in namespace Videos, no detailed information the clients that rented it
- at the *application level*, we go inside the aggregate value and we discover that it it was rented by Client 375657
- at the *application level*, we can execute get(Clients, 375657) to retrieve information about a/the customer that rented video 1234 (thus navigating the customer reference stored inside the video)

Example 3 - Relationships

Find the age(s) of custo) that rented a given vi



```
{"loc": 1234,
  "type":"dvd",
  "rentals": [{"rental": {"rentalDate":"15/10/20
                          "codCli": 375657}
  "title": "pulp fiction",
  "director": "quentin tarantino"
```



```
ctio
ino"}}]
```

REMARKS:

1. Who knows that inside BLOB1 there is the code of the client that rented the video?

And where to look for the client information and how to match?

The application, the data store is completely unaware of that!

2. Since the relationship is not part of the aggregate, navigating it requires a sort of join, no system support for it

3. Since the relationship is not part of the aggregate, navigating it (at application level) requires two distinct data accesses, at two possibly distinct data nodes

ce Videos, no detailed

- aggregate value and we discover 75657
- a e can execute get(Clients, 375657) to retrieve in a/the customer that rented video 1234 (thus nav g the customer reference stored inside the video)

Advanced interaction

-

- Some systems support additional functionality
  - Use of indexes: The data must be indexed first
  - Some kind of additional index (e.g. full text) can be used
  - Example: Riak search

Key-value stores use cases

K-V Stores: Suitable Use Cases

- Storing Web Session Information
  - Every web session is assigned a unique session_id value
  - Everything about the session can be stored by a single PUT request or retrieved using a single GET
  - Fast, everything is stored in a single object

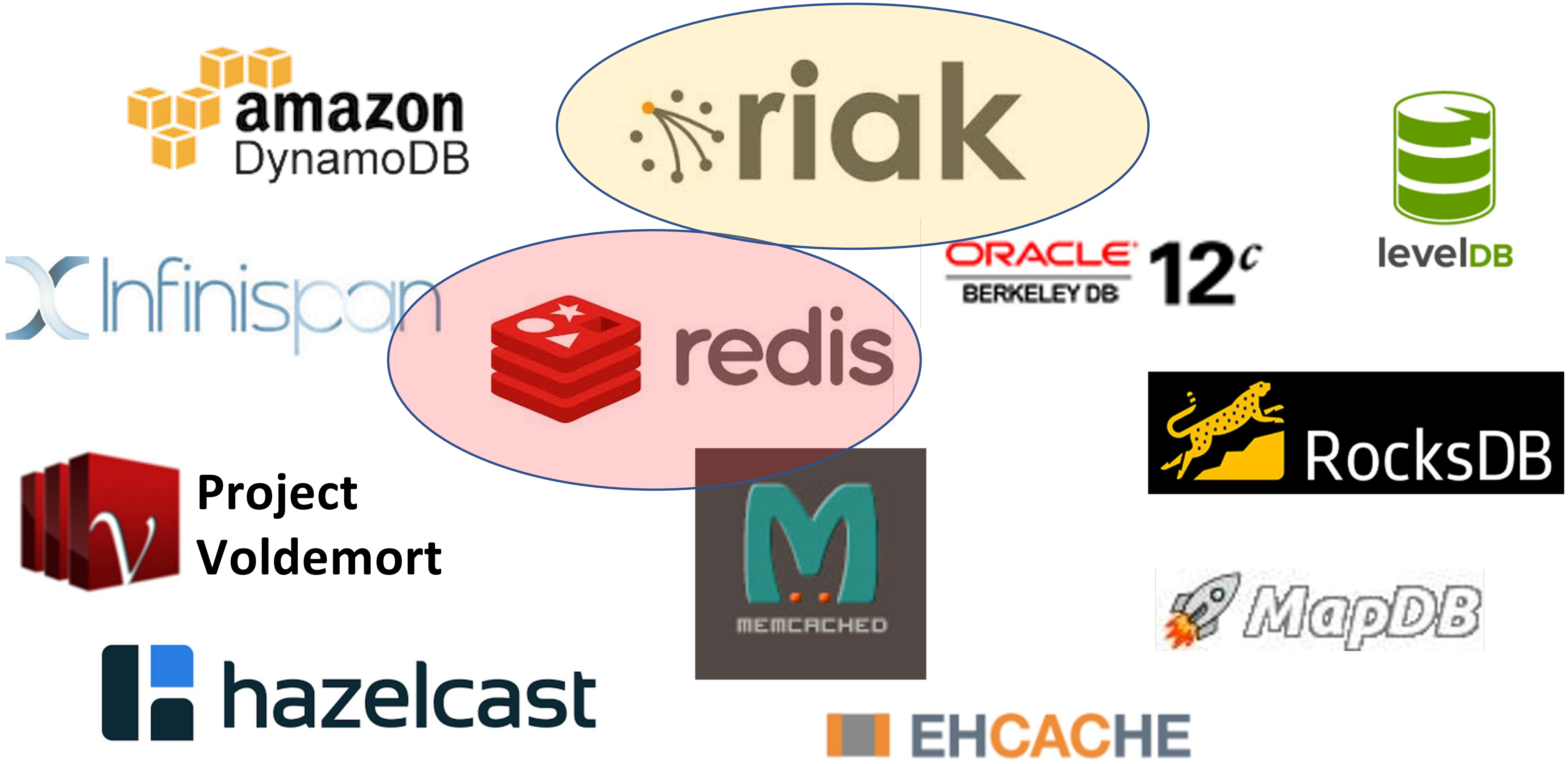
- User Profiles, Preferences
  - Every user has a unique user_id/user_name + preferences (language, time zone, design, access rights, …)
  - As in the previous case: Fast, single object, single GET/PUT

- Shopping Cart Data
  - Similar to the previous cases

K-V Stores: When Not to Use

- Relationships among Data
  - Relationships between different sets of data
    - Some key-value stores (Riak) provide link-walking features
- Multi-operation Transactions
  - Saving multiple keys
    - Failure to save any of them → revert or roll back the rest of the operations
- Query by Data
  - Search the keys based on something found in the value part
    - Additional indexes needed (some stores provide them)
- Operations by Key Sets
  - Operations are limited to one key at a time
    - No way to operate upon multiple keys at the same time

Popular key-value data stores

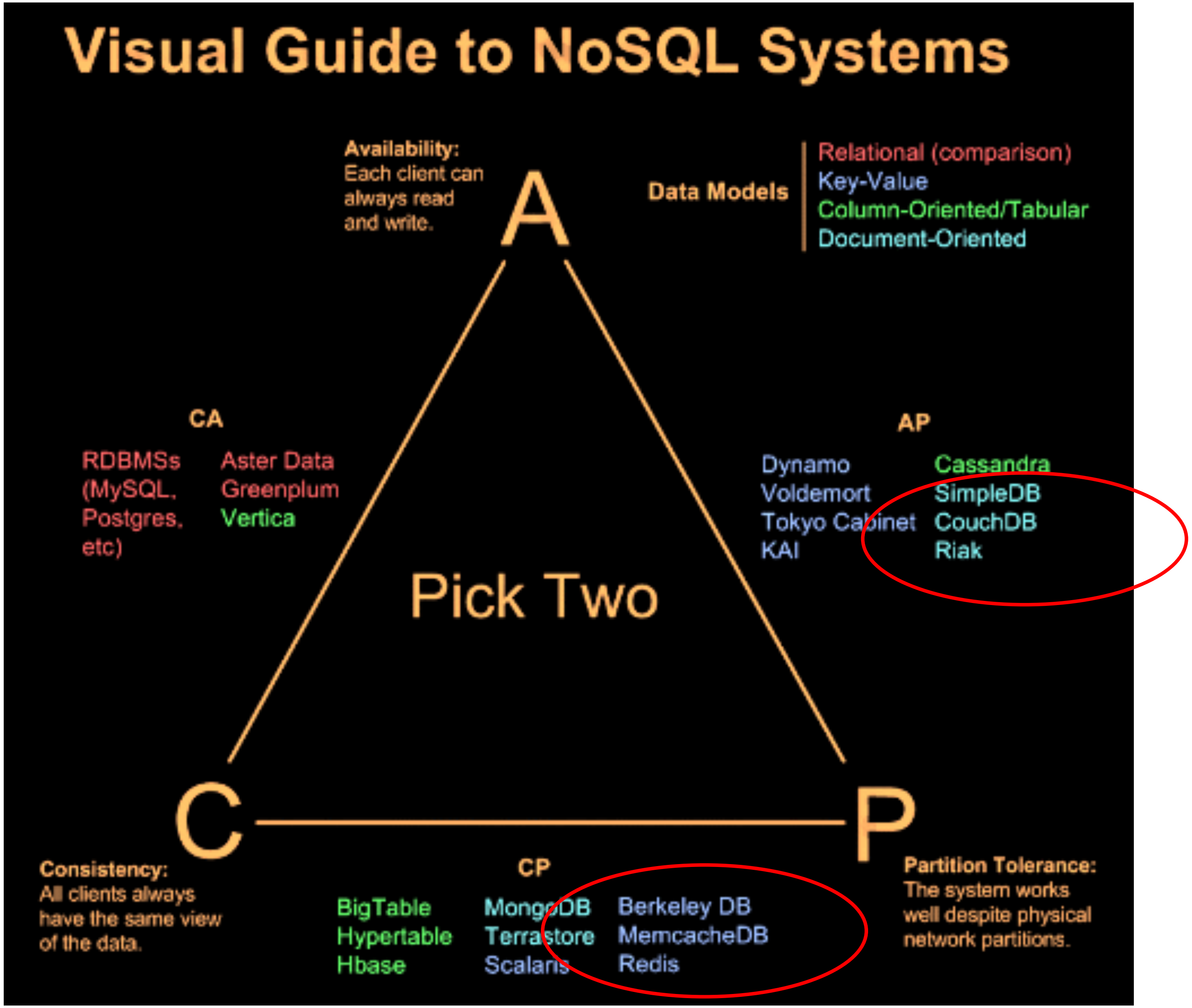


Ranked list: http://db-engines.com/en/ranking/key-value+store

Key-value stores are quite diverse

- Dozens of key-value stores – are all of them the same?
  - Embedded local storages
    - LevelDB
      - Local storage for many systems, Log-structured Merge Tree
  - Distributed key-value Stores
    - Riak, Infinispan
  - Memory caches
    - Redis, Memcached

CAP theorem: key-value stores


Visual Guide to NoSQL Systems
Availability:
Each client can always read and write.
A
Data Models
Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick Two
C
P
Consistency:
All clients always have the same view of the data.
CP
BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance:
The system works well despite physical network partitions.