Aggregate-oriented logical design

Relational database design

- Conceptual design
  - From the requirements to a conceptual schema
  - *Conceptual model*: entity-relationship model
- Logical design
  - From a conceptual schema to a logical schema
  - *Logical model*: relational model
  - The logical schema can then be directly created in an RDBMS
- Physical design
  - Optimizing the storage of the database on disk
  - Indexing, clustering,…

Aggregate-oriented design

- Conceptual design
  - From the requirements to a conceptual schema + workload
  - *Conceptual model*: entity-relationship model

- Logical design
  - From a conceptual schema + workload to a logical schema
  - *Logical model*: one specific aggregate-oriented model
  - Often useful to start by designing a meta-logical schema, based on a meta-notation corresponding to JSON-like schema
  - Given a NoSQL system, the meta-logical schema has to be translated into the aggregate-oriented model provided by the system

- Physical design
  - Optimizing the storage of the database on disk
  - Partitioning (always), indexes (sometimes)

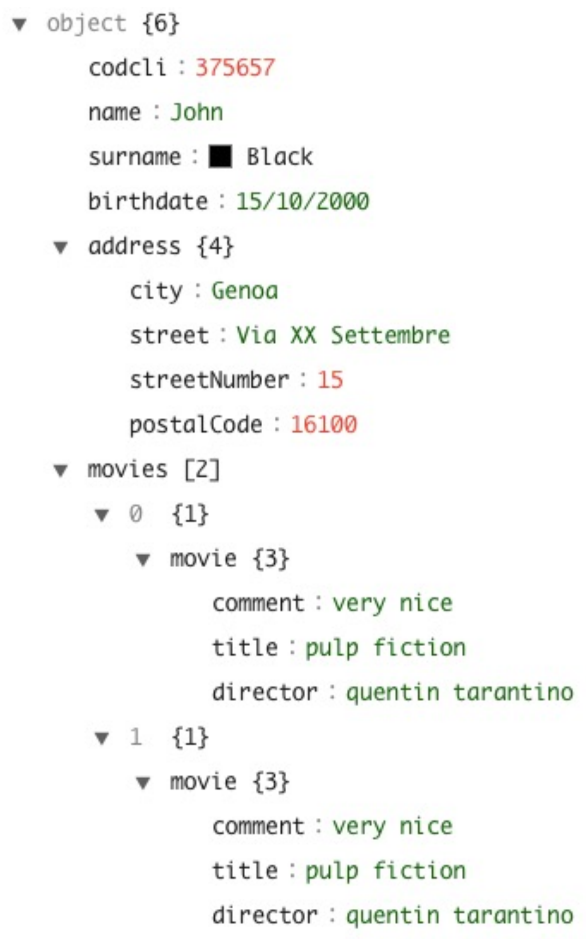
Aggregate-oriented logical design: principles

1. Minimize the number of joins by minimizing the number of collections
2. Reduce redundancy by using complex attributes (lists, sets)
3. Selections over simple attributes are more efficient than selections over complex attributes (sets, lists)
    - In some systems, selections over complex attributes are not possible

Aggregate

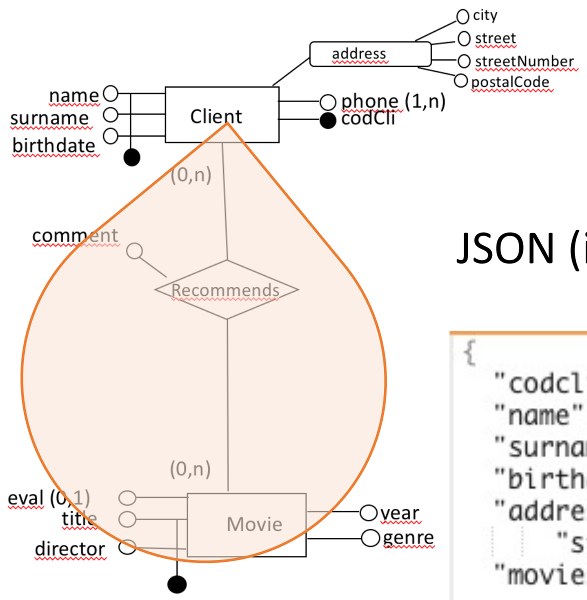
- Taking a reference workload into account, we extend the attributes of an entity E with the attributes of other entities associated with E or of associations involving E
- We «encapsulate» in E properties related to other entities or associations to facilitate their retrieval, based on the operations contained in the workload

JSON – tree view



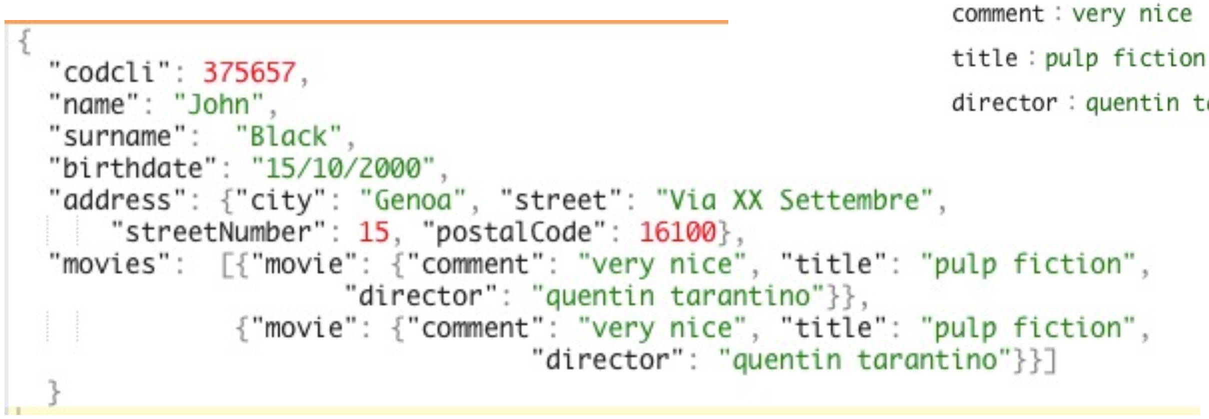
```
▼ object {6}
    codcli : 375657
    name : John
    surname : ■ Black
    birthdate : 15/10/2000
  ▼ address {4}
      city : Genoa
      street : Via XX Settembre
      streetNumber : 15
      postalCode : 16100
  ▼ movies [2]
    ▼ 0  {1}
      ▼ movie {3}
          comment : very nice
          title : pulp fiction
          director : quentin tarantino
    ▼ 1  {1}
      ▼ movie {3}
          comment : very nice
          title : pulp fiction
          director : quentin tarantino
```

ER schema



{} object (name-value pair)
[] array

JSON (instance level)



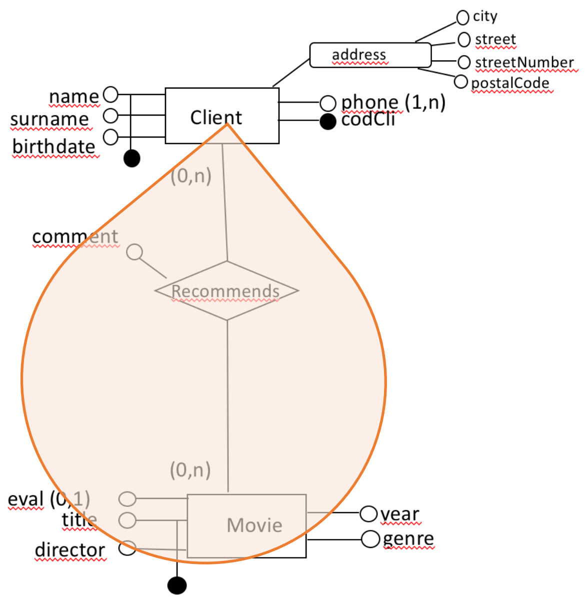
```
{
  "codcli": 375657,
  "name": "John",
  "surname":  "Black",
  "birthdate": "15/10/2000",
  "address": {"city": "Genoa", "street": "Via XX Settembre",
       "streetNumber": 15, "postalCode": 16100},
  "movies":  [{"movie": {"comment": "very nice", "title": "pulp fiction",
                    "director":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

Aggregate schema

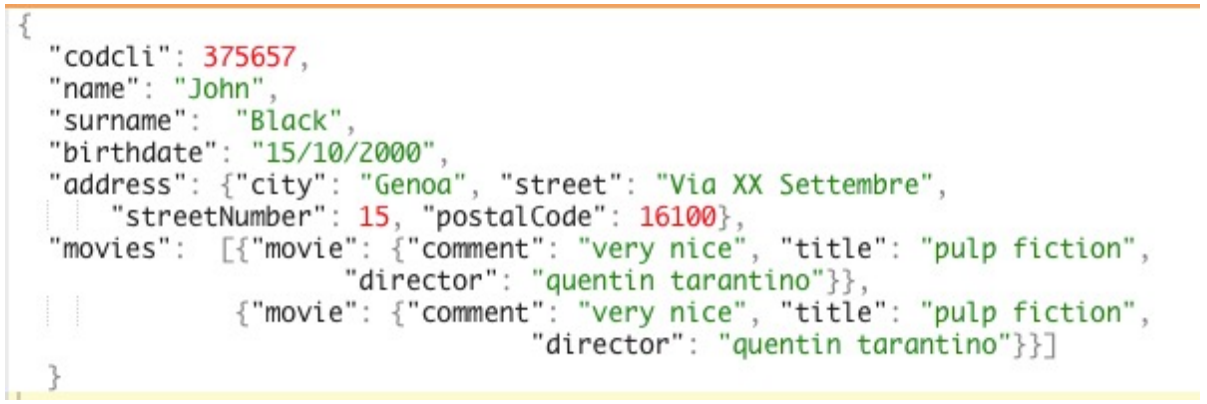
{} object (name-value pair)
[] array

- We rely on a JSON schema meta-notation

ER schema



JSON (instance level)



```
{
  "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 fiction",
                    "director": "quentin tarantino"}},
                {"movie": {"comment": "very nice", "title": "pulp fiction",
                           "director": "quentin tarantino"}}]
 }
```

JSON schema meta-notation (semistructured schema level)

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

We underline all sets of attributes that represent identifiers

JSON schema

```
{
  "type": "object",
  "title": "Client",
  "description": "",
  "properties": {
      "codCli": { "type": "string", "description": "" },
      "name": { "type": "string", "description": "" },
      "surname": { "type": "string", "description": "" },
      "birthdate": { "type": "string", "description": "" },
      "address": {
                    "type": "object",
                    "properties": {
                           "city": { "type": "string", "description": "" },
                           "street": { "type": "string", "description": "" },
                           "streetNumber": { "price": "string", "description": "" },
                           "postalCode": { "type": "string", "description": "" }
                       },
           "required": ["city", "street", "streetNumber", "postalCode"]
             },
     "movies": {
          "type": "array",
          "description": "",
          "items": {
                    "type": "object",
                    "title": "movie",
                    "description": "",
                    "properties": {
                           "title": { "type": "string", "description": "" },
                           "director": { "type": "string", "description": "" },
                         "comment": { "type": "string", "description": "" }
                               },
                   "required": ["title", "director", "comment"]
                   },
             "minItems": 0
             }
         },
   "required": ["codCli", "name", "surname", "birthDate", "address"],
   " id": ["codCli", ["name", "surname", "birthdate"]]
}
```

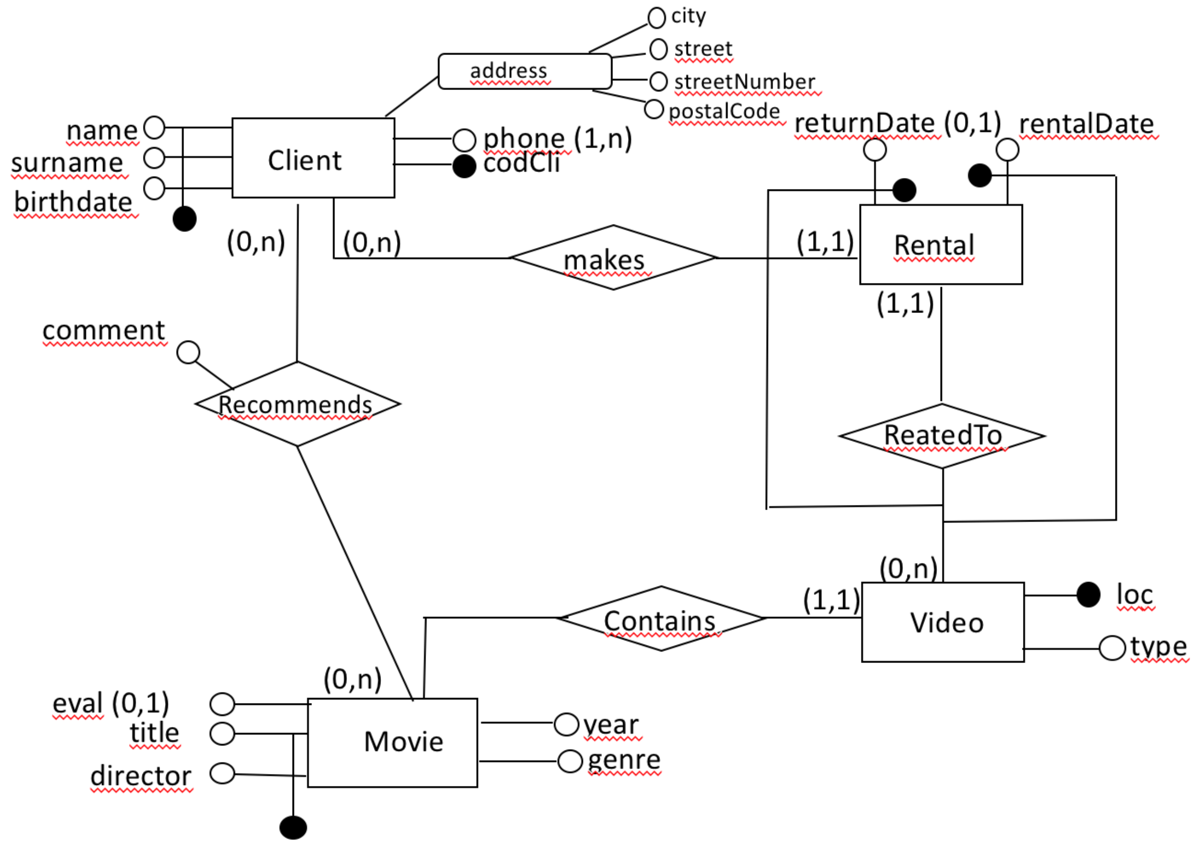
JSON schema meta-notation

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

Methodology

- Input
  - ER schema
  - Workload
- Step 1
  - Each query in the workload is modeled in a formal and non ambiguous way
- Step 2
  - The ER schema is annotated with query information
- Step 3
  - The aggregate-oriented logical schema is generated starting from the annotated ER schema
- Output
  - The aggregate-oriented logical schema

Workload

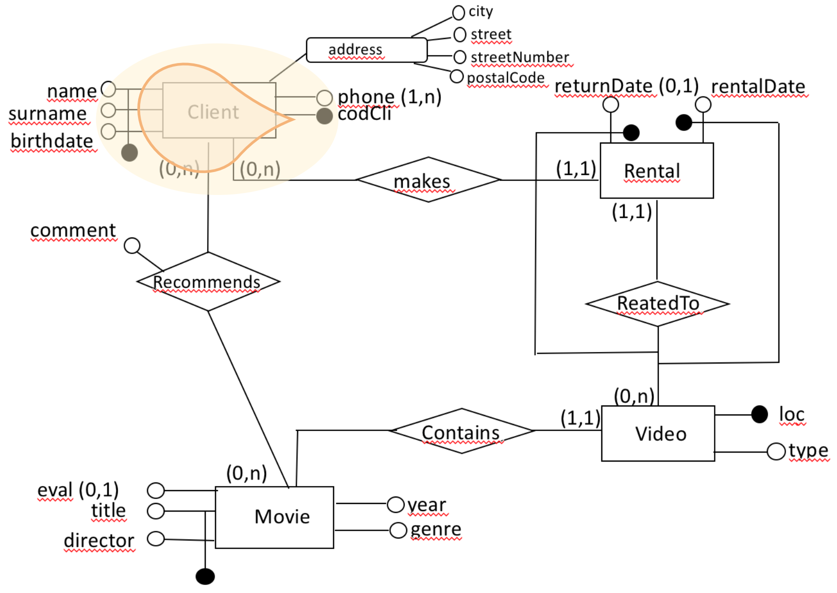


- **Q1.** Average age of clients
- **Q2.** Name and surname of clients and related recommended movies
- **Q3.** Genre and year of the movies and their related recommendations, together with the name and the surname of the client who made them
- **Q4.** Name and surname of clients who recommended the movie 'pulp fiction' by 'quentin tarantino'
- **Q5.** Given a movie, all information of videos that contain it
- **Q6.** Videos of type 'DVD', rented from a certain date
- **Q7.** The videos of type 'VHS' containing the movie 'pulp fiction' by 'quentin tarantino' and the clients that rented them

Step 1: query modeling

- For each query Q in the workload, determine
  - The entity E to be used as aggregate
  - The set of entities LS used for selecting data to be returned as query result (= FROM and WHERE clauses of an SQL query), the attributes used in the selections, and the path connecting each of them to E
    - If LS is not empty, E must be selected from LS entities
  - The set of entities LP used for projections (= SELECT clause of an SQL query), the attributes used in the projections, and the path connecting each of them to E
  - If you can choose, better if E appears at the (0,1) or (1,1) side of an association connecting selection/projection entities [as we will see, this limits selection conditions over nested attributes]
- Q → Q (E, LS, LP)

Case 1: no selection entity, one projection entity (projection over some entity attributes)

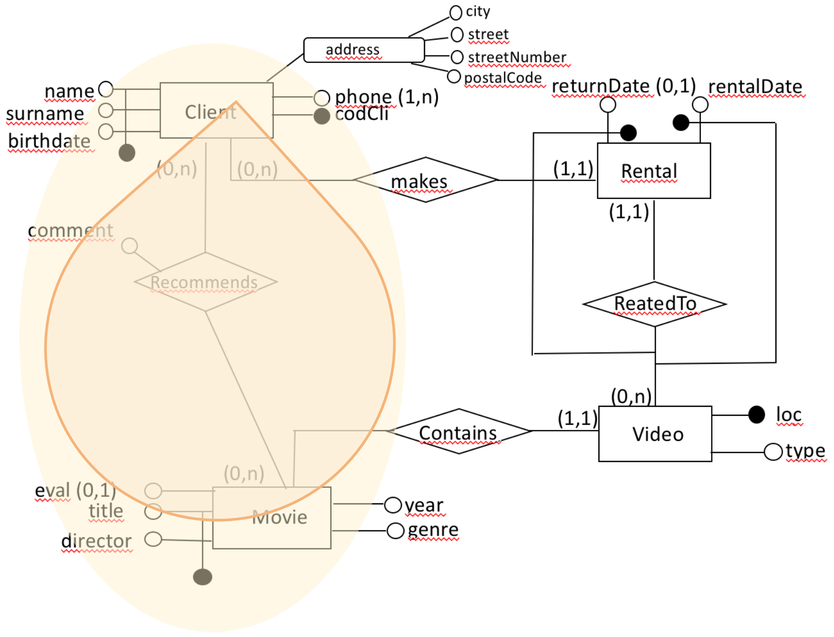


- Q1. Average age of clients

- E = Client
- LS = [ ]
- LP = [Client (birthdate) _!]

- Q1 → Q1(Client,
  [ ],
  [Client (birthdate) _!])

Case 2: no selection entity, two projection entities (projection over some entity attributes)

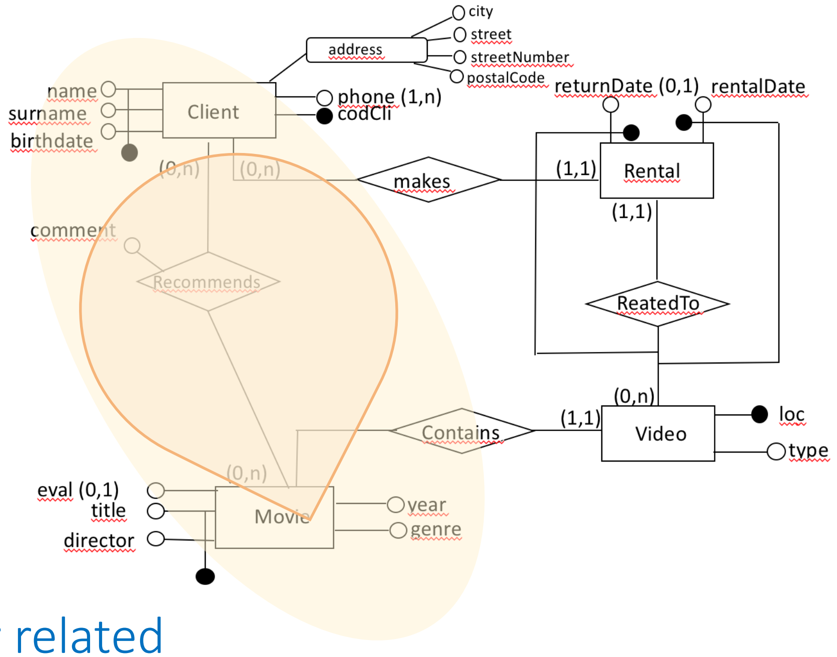


- **Q2.** Name and surname of clients and related recommended movies

- E = Client
- LS = [ ]
- LP = [Client(name, surname)_!, Movie(title, director)_R]

- Q2 (Client,
  [ ],
  [Client(name, surname)_!, Movie(title, director)_R])

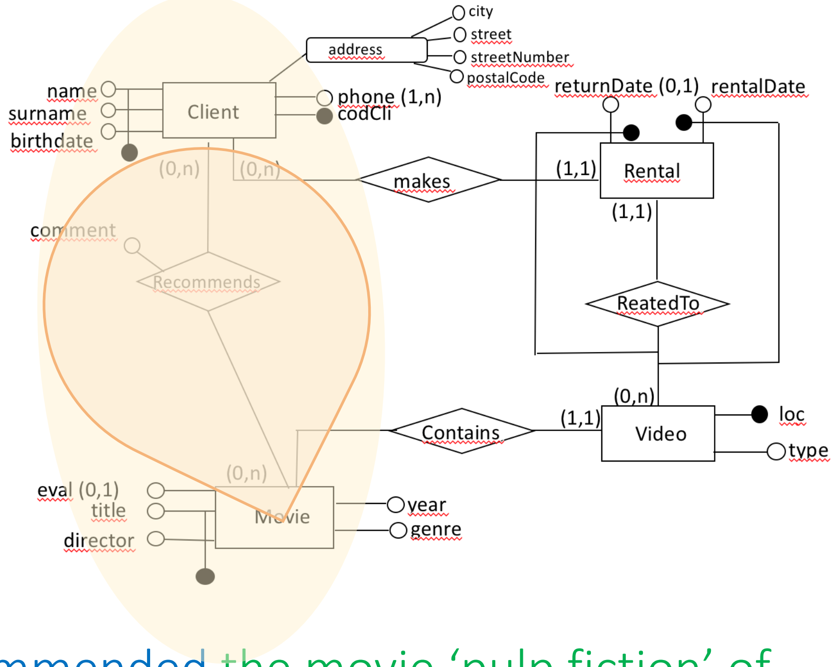
Case 3: no selection entities, two projection entities (projection over entity and association attributes)



- Q3. Genre and year of the movies and their related recommentations, together with the name and the surname of the client who made them
- E = Movie
- LS = [ ]
- LP = [Movie(genre, year)_!, Client(name, surname)_R(comment)]

- Q3 (Movie,
  [ ],
  [Movie(genre, year)_!, Client(name, surname)_R(comment)])

Case 4: one selection entity, one projection entity (projection over some entity attributes)

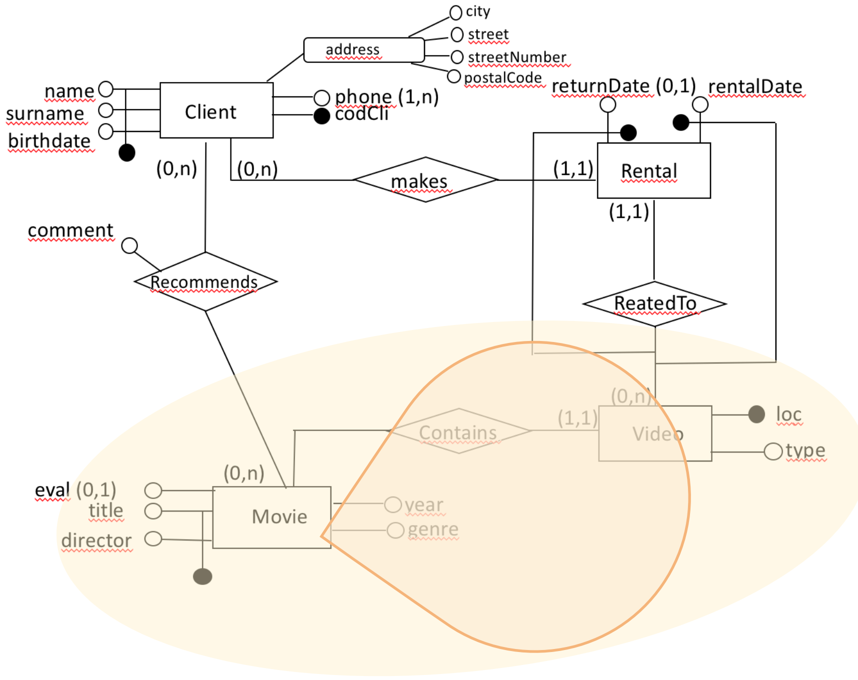


- **Q4.** Name and surname of clients that recommended the movie 'pulp fiction' of 'quentin tarantino'

- E = Movie
- LS = [Movie(title, director)_!]
- LP = [Client(name, surname)_R]

- Q4 (Movie,
  [Movie(title, director)_!],
  [Client(Name, Surname)_R])

Case 5: one selection entity, one projection entity (on all the attributes)

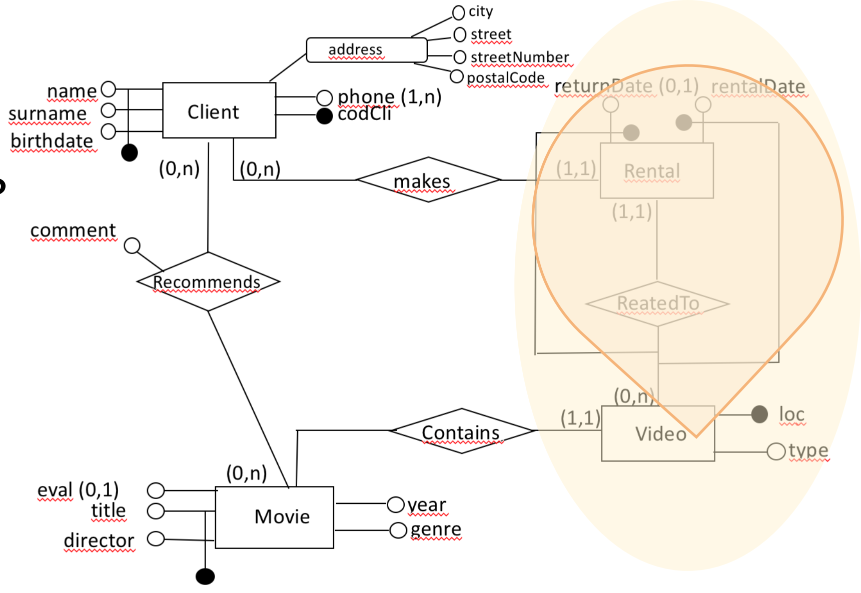


- Q5. Given a movie, all information of videos that contain it

- E = Movie
- LS = [Movie(title, director)_!]
- LP = [Video_C]

- Q5 (Movie,
  [Movie(title, director)_!],
  [Video_C])

Case 6: two selection entities, one projection entity (over some entity attributes)

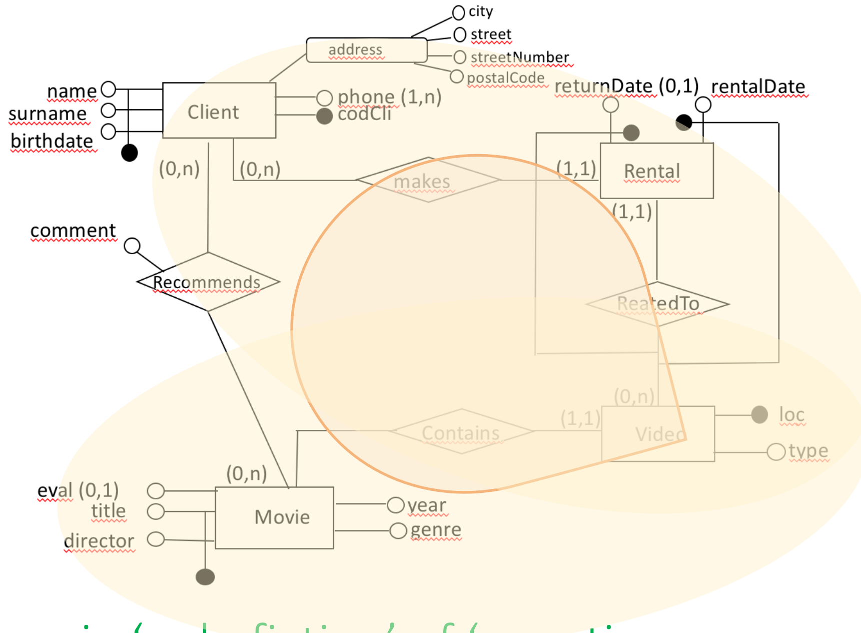


- **Q6.** The videos of type 'dvd', rented from a given date

- E = Video
- LS = [Video(type)_!, Rental(rentalDate)_Rt]
- LP = [Video(loc)_!]

- Q6 (Video,
  [Video(type)_!, Rental(rentalDate)_Rt],
  [Video(loc)_!])

Case 7: two selection entities, two projection entities



- **Q7.** The videos of type 'vhs' containing the movie 'pulp fiction' of 'quentin tarantino' and the clients that rented them
- E = Video
- LS = [Video(type)_!, Movie(title, director)_C]
- LP = [Video(loc)_!, Client(codCli)_MRT]

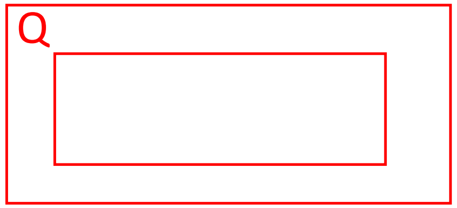
- Q7 (Video,

  [Video(type)_!, Movie(title, director)_C]

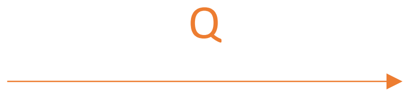
  [Video(loc)_!, Client(CodCli)_MRt])

Step 2: annotation of the ER schema

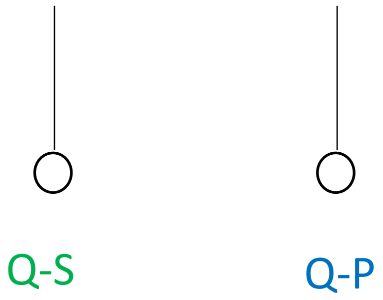
- The ER schema is annotated with information from each query Q (E, LS, LP)



Aggregation entity E

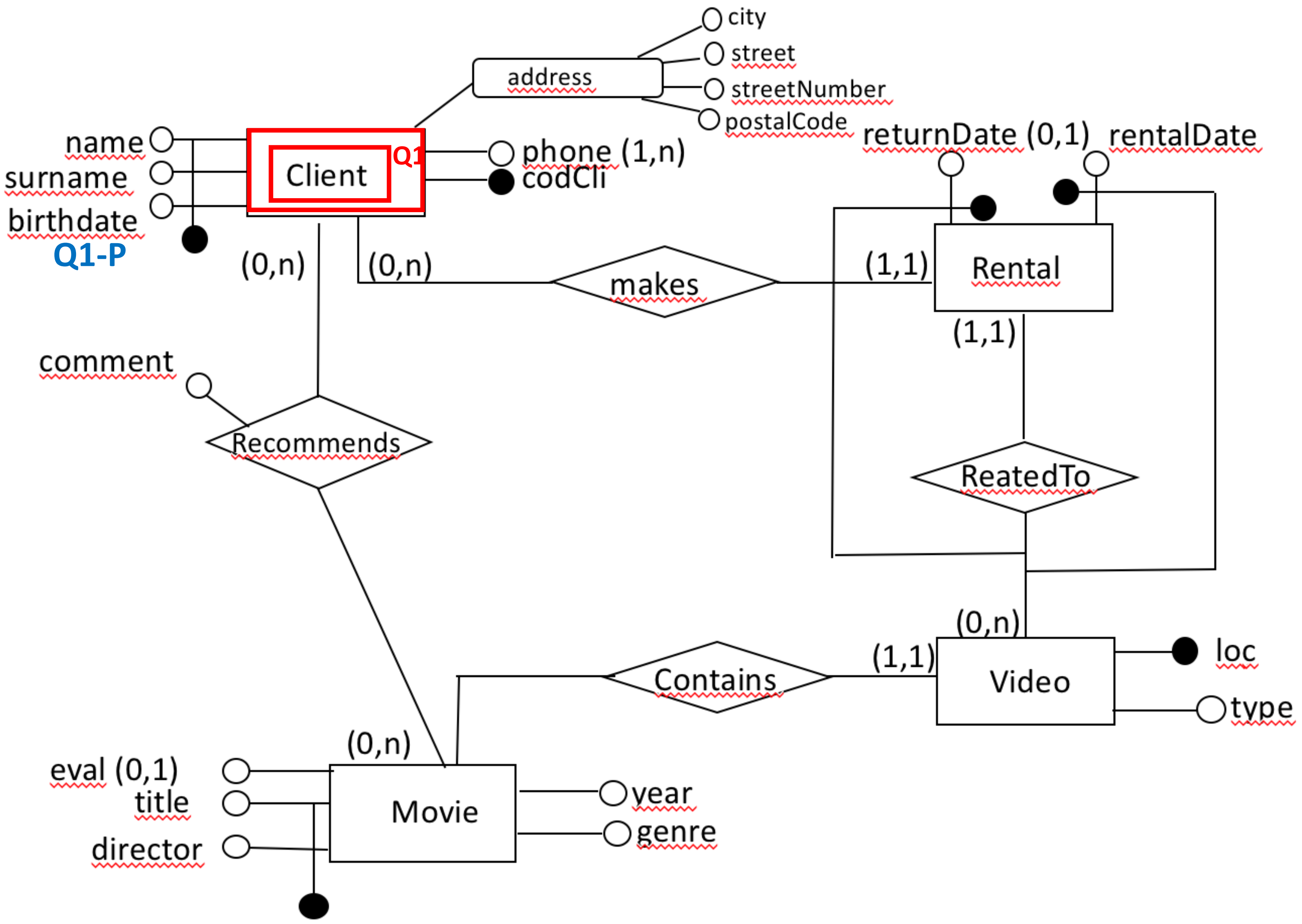


Navigation arrows on association edges, described in LS and LP, towards the aggregation entity

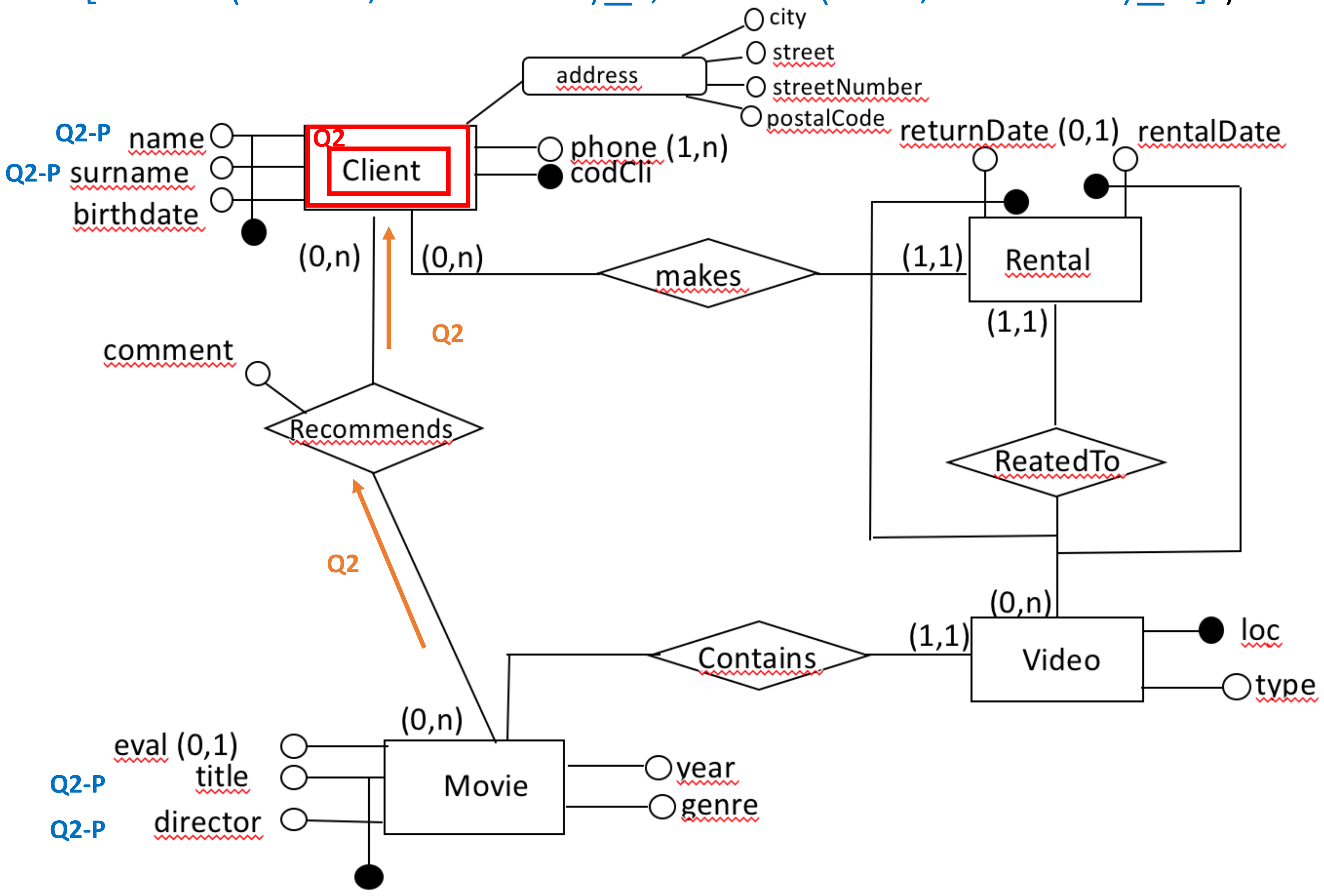


Reference to query Q on each attribute involved in
selection LS: Q-S
projection LP: Q-P

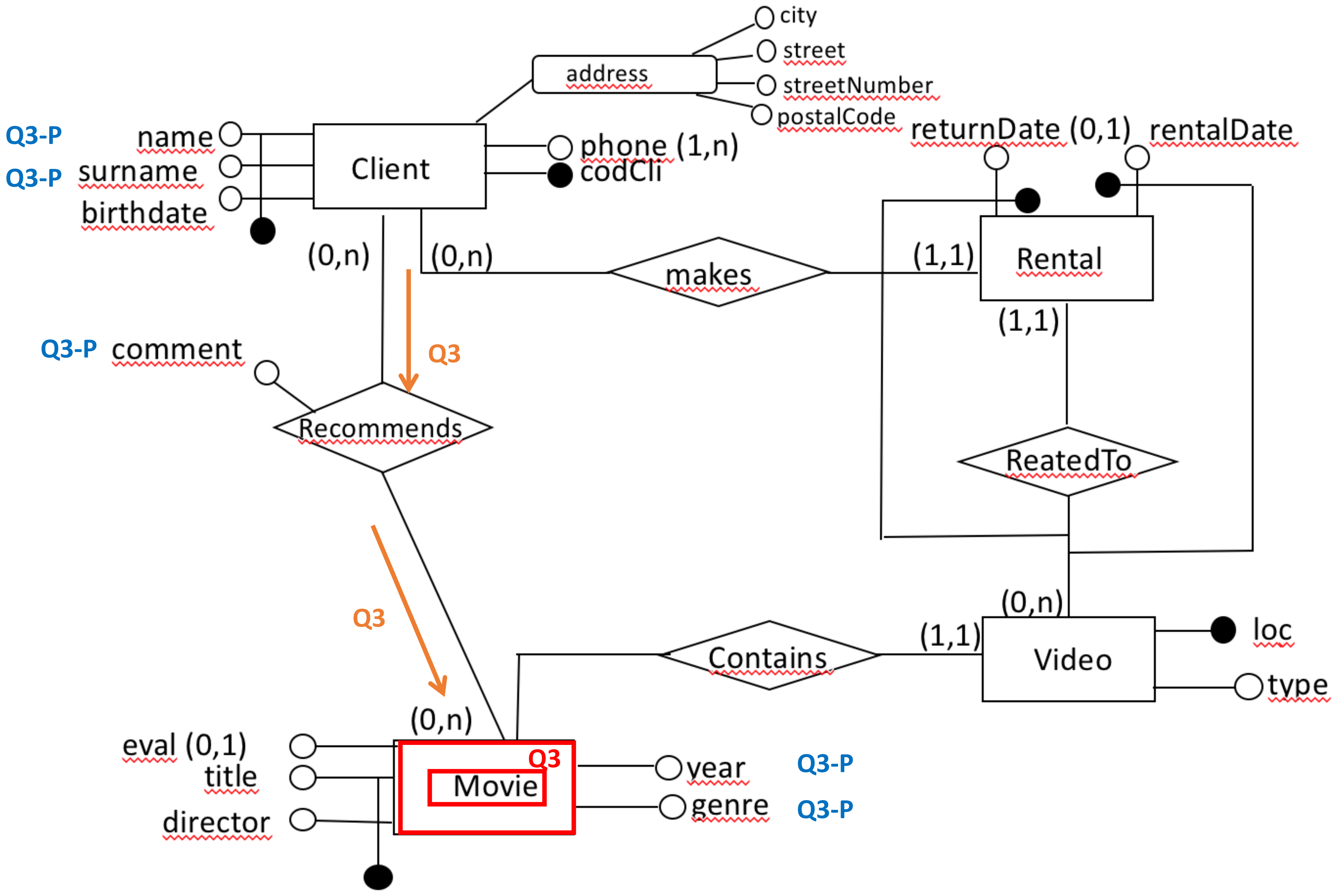
Q1(Client, [ ], [Client(birthdate)_!])



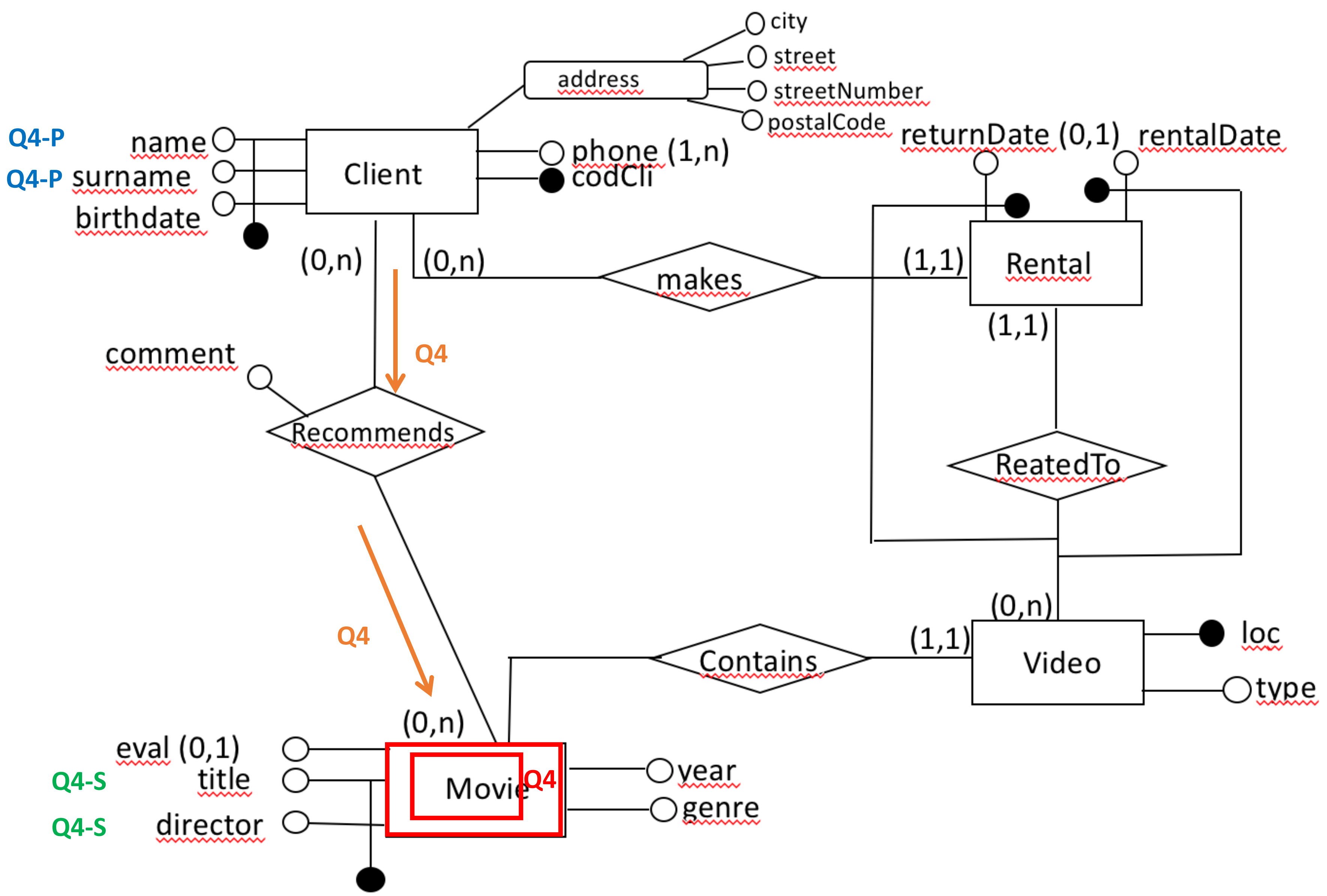
Q2 (Client, [ ],
[Client(name, surname)_!, Movie(title, director)_R])



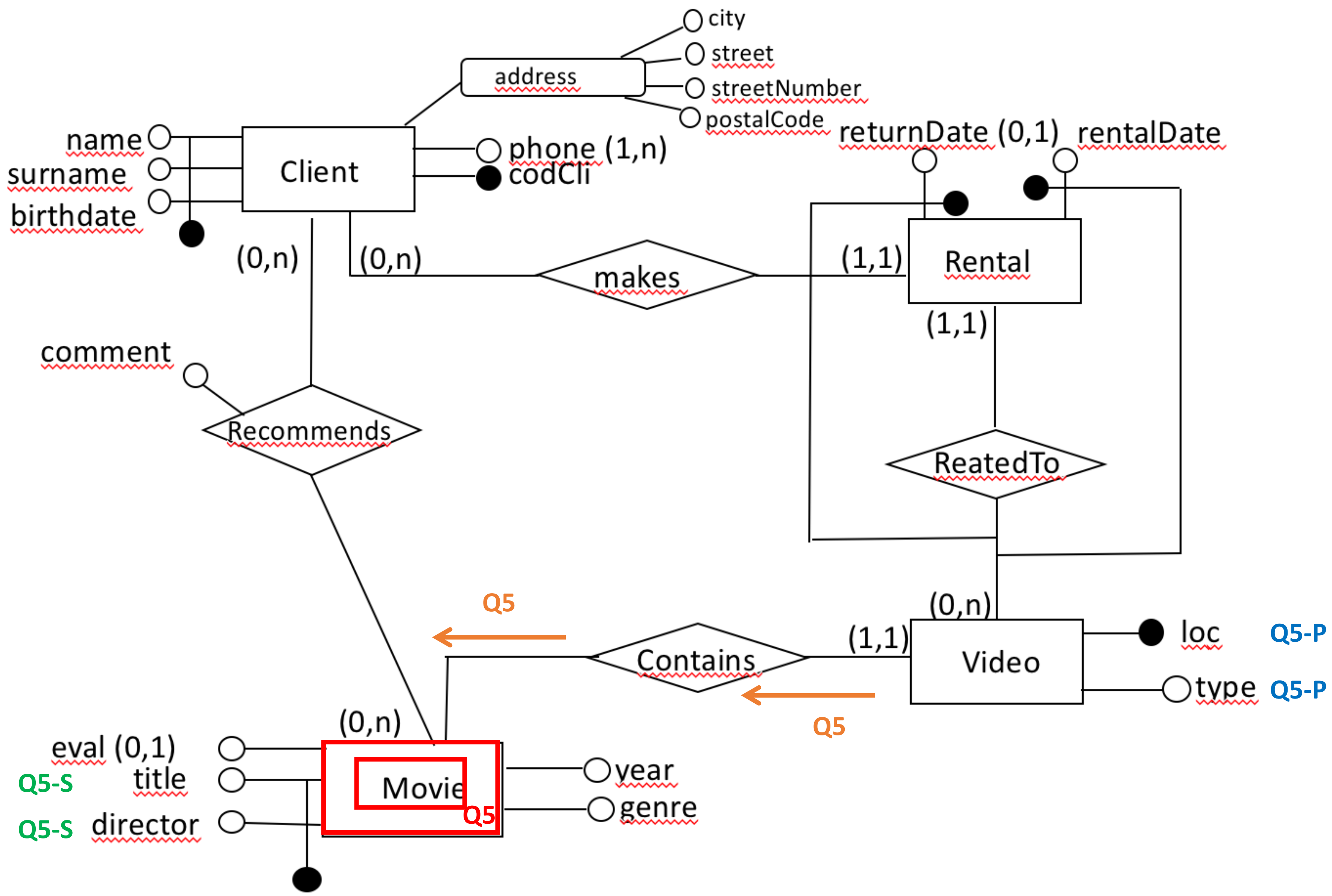
Q3 (Movie, [ ],
[Movie(genre, year)_!, Client(name, surname)_R(comment)])



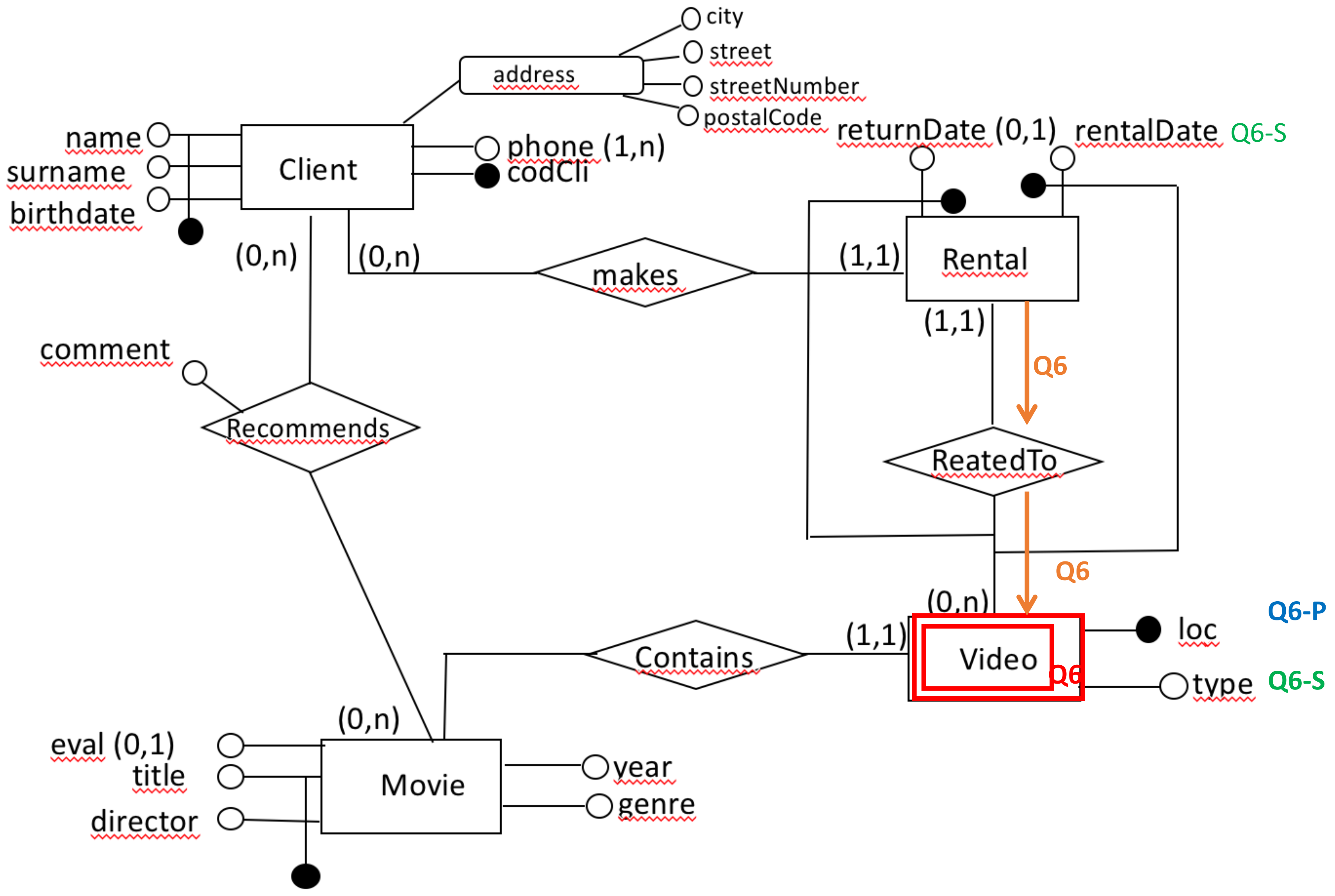
Q4 (Movie, [Movie(title, director)_!], [Client(name, surname)_R])



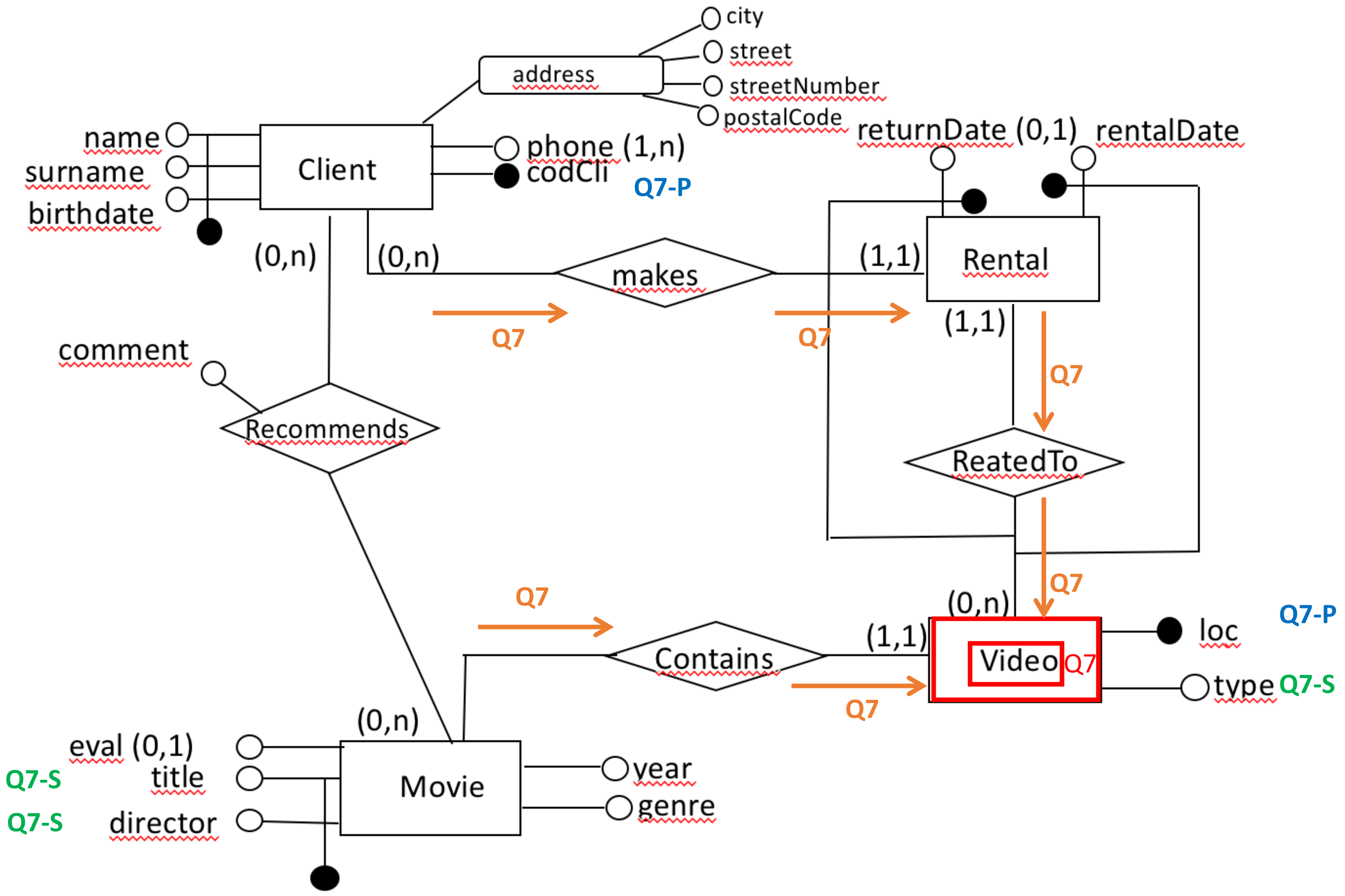
Q5 (Movie, [Movie(title, director)_!], [Video_C])



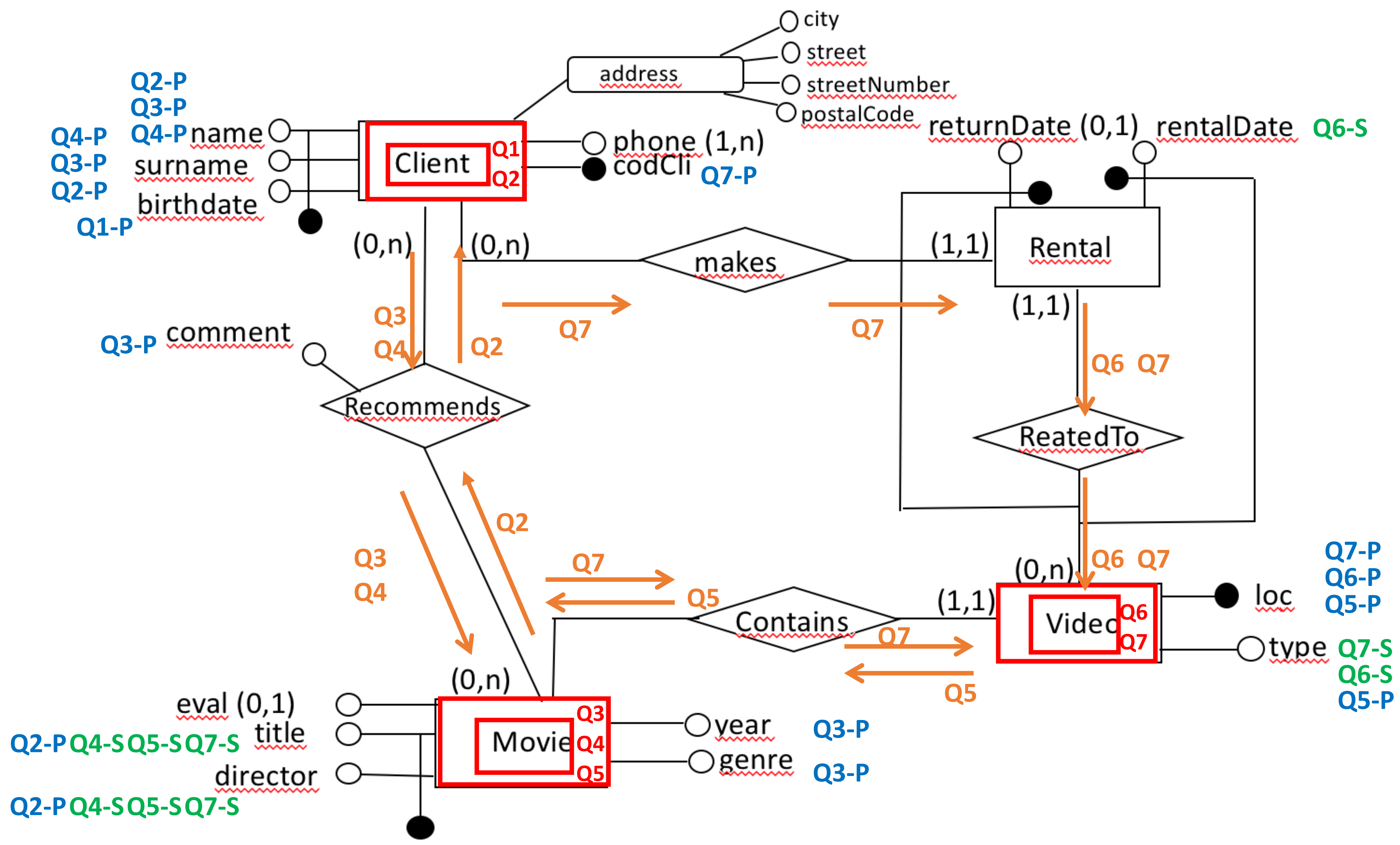
Q6 (Video, [Video(type)_!, Rental(rentalDate)_Rt], [Video(loc)_!])



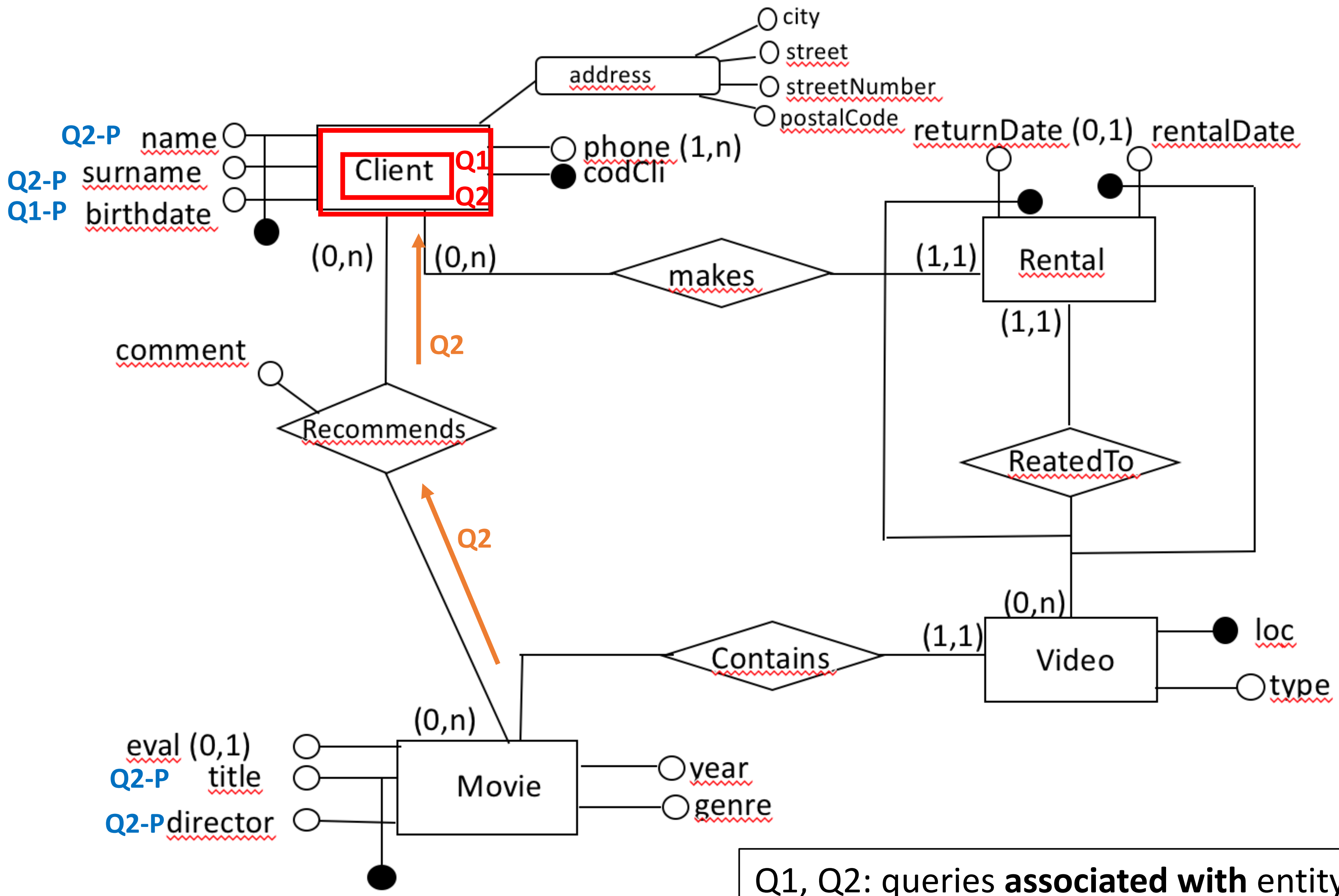
Q7 (Video, [Video(type)_!, Movie(title, director)_C]
[Video(loc)_!, Client(codCli)_MRt])



Final annotated ER schema

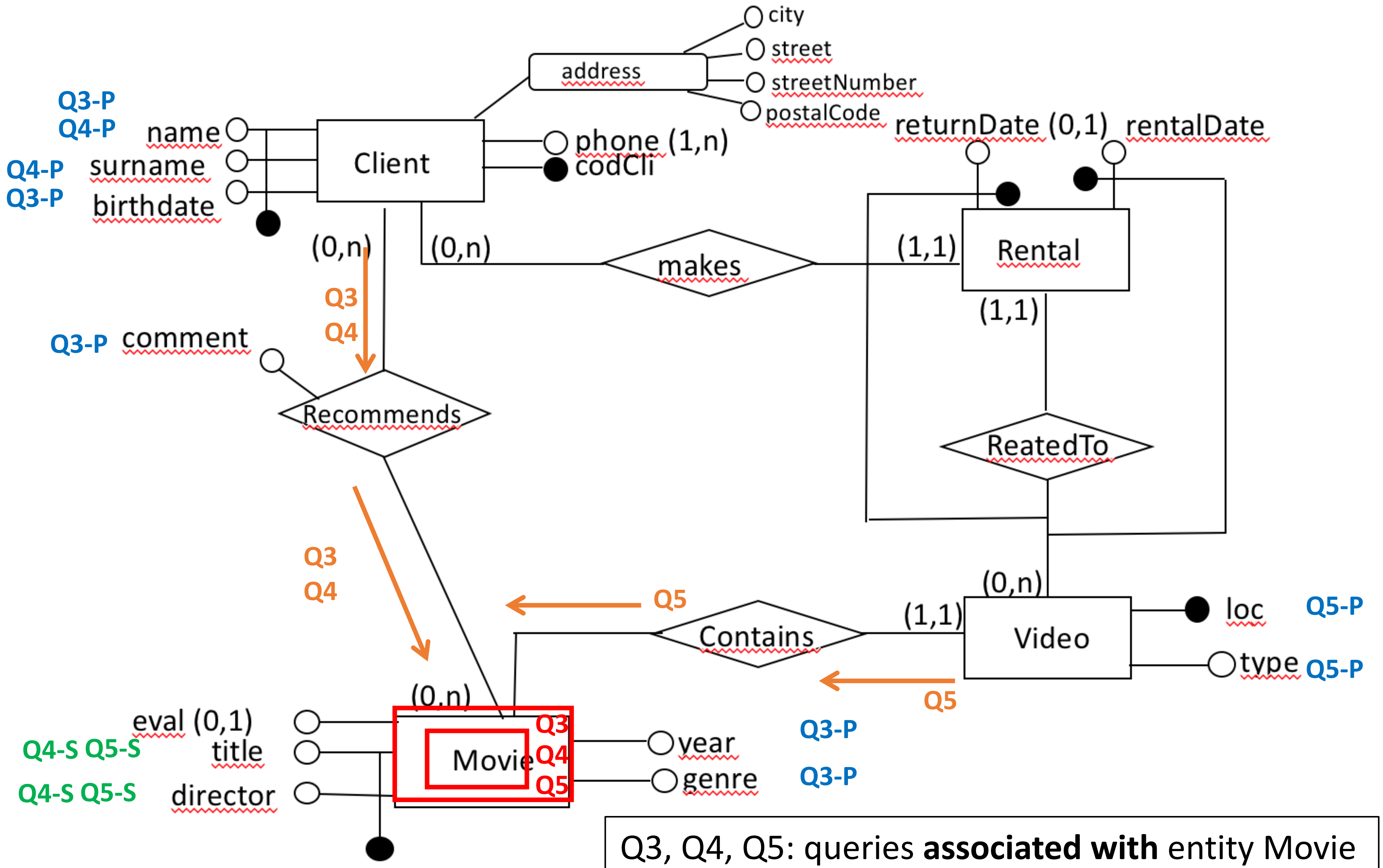


Final annotated ER schema



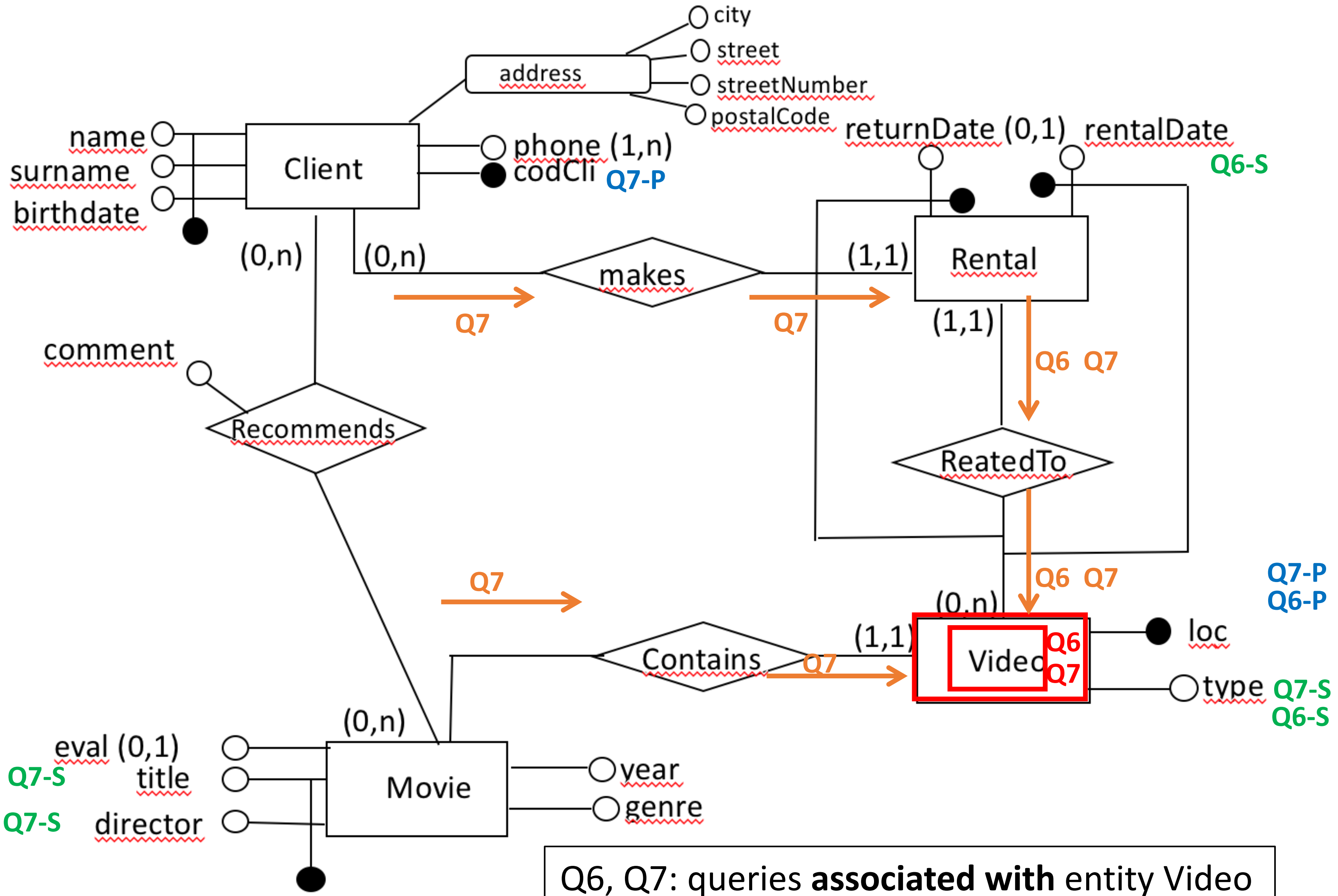
Q1, Q2: queries **associated with** entity Client

Final annotated ER schema



Q3, Q4, Q5: queries **associated with** entity Movie

Final annotated ER schema

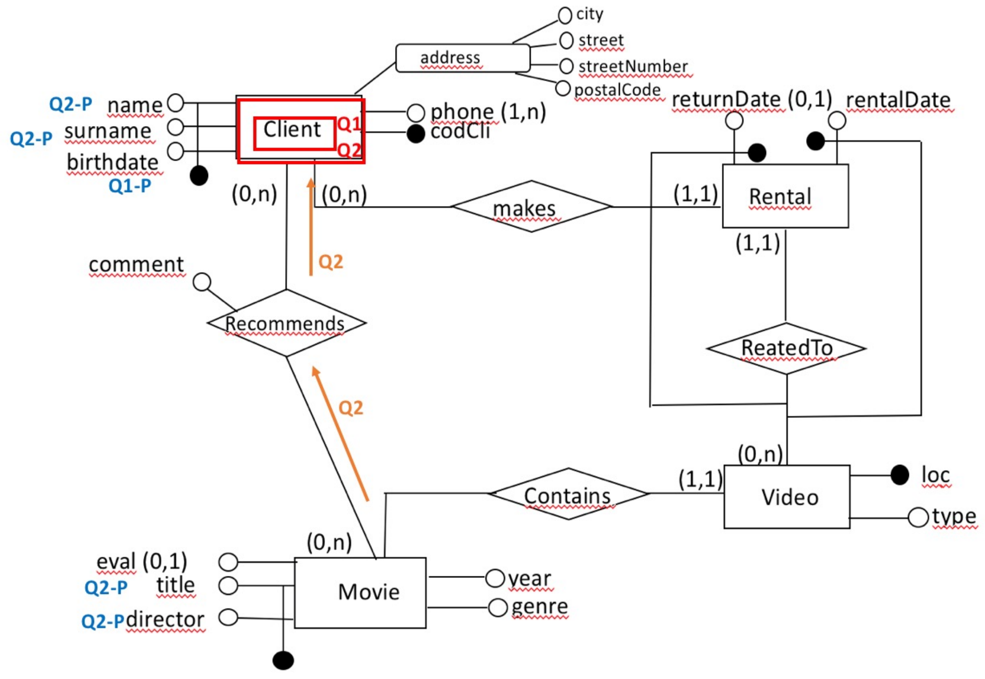


Q6, Q7: queries **associated with** entity Video

Step 3: translation into a JSON meta-notation

- Create one collection E for each aggregation entity E
- Aggregates inside collection E are characterized by
    - All attributes of entity E annotated with queries associated with E (= their identifier is located inside E)
        - Represented as simple attributes
    - Further attributes corresponding to entities and associations located on paths annotated with one query associated with E
        - Represented as simple or complex attributes depending on the cardinality of associations included in the considered path
    - An id for the aggregate, corresponding to one of the identifiers for E in the ER schema

Aggregation entity Client

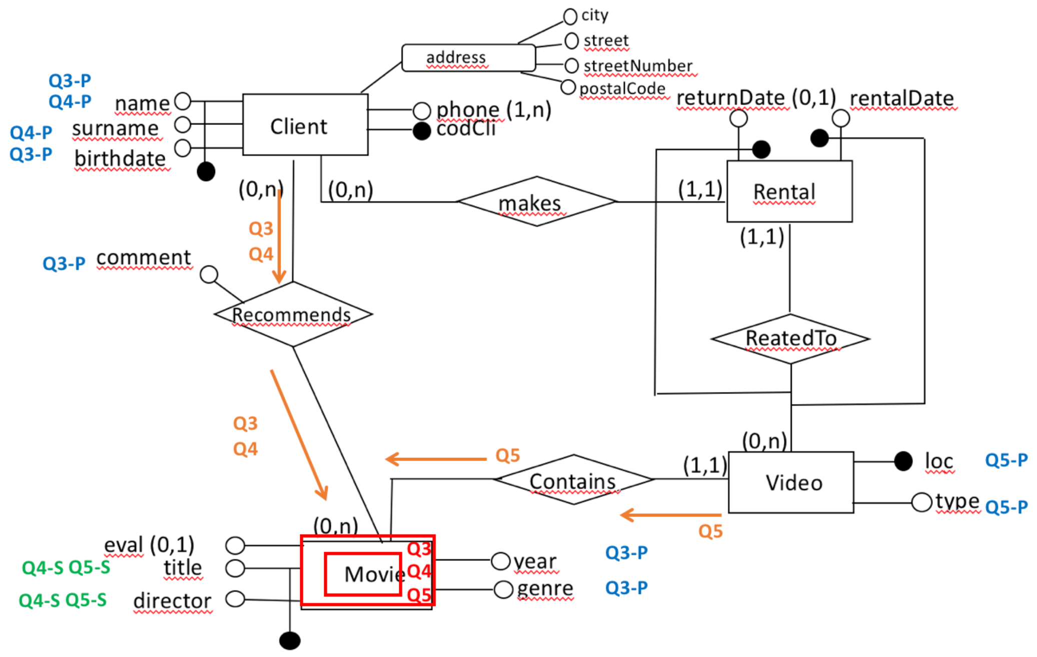


- Create collection Client
- Queries associated with Client
  - Q1, Q2
- Add all the attributes of Client annotated by Q1 or Q2
  - name, surname, birthdate, …
- Now Consider Q1
  - No path entering in Client is annotated by Q1 → no further attributed added

client:
{name, surname, birthdate,
recommends: [{movie: {title, director}}]
}

- Now consider Q2
  - One single path of length 1 (= one association) annotated with Q2 : Client – Recommends - Movie
  - Cardinality constraint for Client in association Recommend: **(0,n)**
  - Add to Client a **set-based attribute** containing tuples composed of attributes of Recommends and Movie annotated with Q2
  - Client (name, surname, birthdate, **recommends: [{movie: {title, director}}**)

Aggregation entity Movie



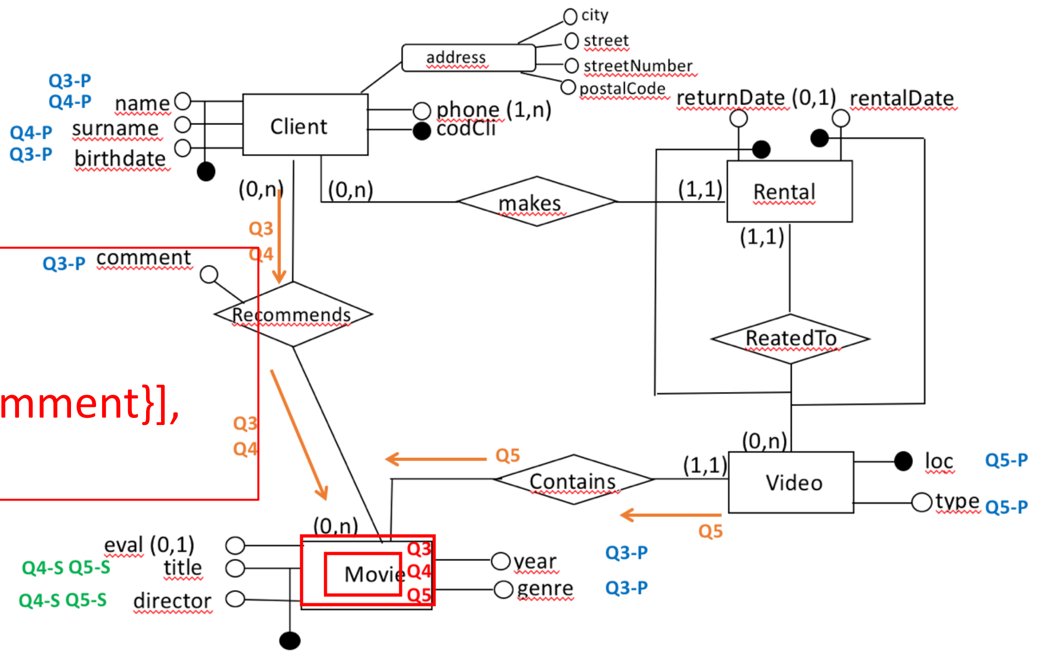
- Create collection Movie
- Queries associated Movie
  - Q3, Q4, Q5
- Add all the attributes of Movie annotated by Q3, Q4, o Q5
  - {**title, director, year, genre**, …
- Now consider Q3
  - One single path of length 1 (= one association) annotated with Q3: Client – Recommends - Movie
  - Cardinality constraint for Movie in association Recommend: **(0,n)**
  - Add to Movie a **set-based attribute** containing tuples composed of attributes of Recommends and Client annotated with Q3
  - movie:

    {title, director, year, genre,

    recommended_by: [{name, surname, comment}],…

Aggregation entity Movie

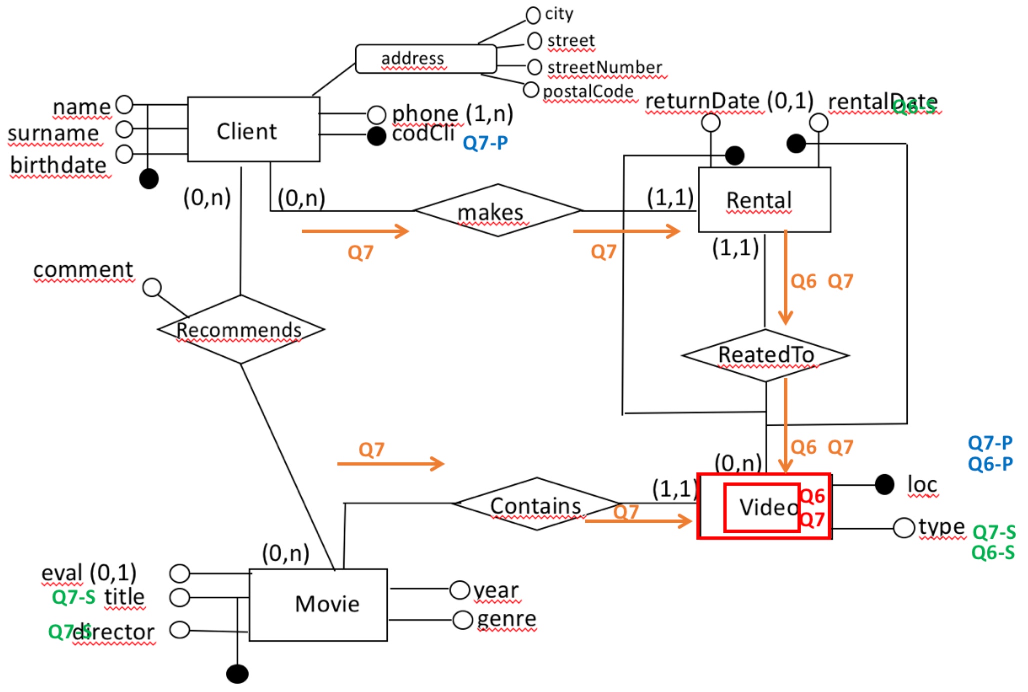
movie:
{<u>title, director</u>, year, genre,
recommended_by: [{name, surname, comment}],
contained_in: [{video: {loc, type}}]}



- Now consider Q4
  - One single path of length 1 annotated with Q4: Client – Recommends - Movie
  - All the attributes annotated with Q4 are also annotated with Q3, thus they have already been inserted in the schema
  - Nothing change
- Now consider Q5
  - One single path of length 1 annotated with Q5: Video – Contains – Movie
  - Cardinality constraint for Movie in association Contains: **(0,n)**
  - Add to Movie one **set-based attribute** containing tuples composed of attributes of Contains and Video annotated with Q5
  - movie:

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

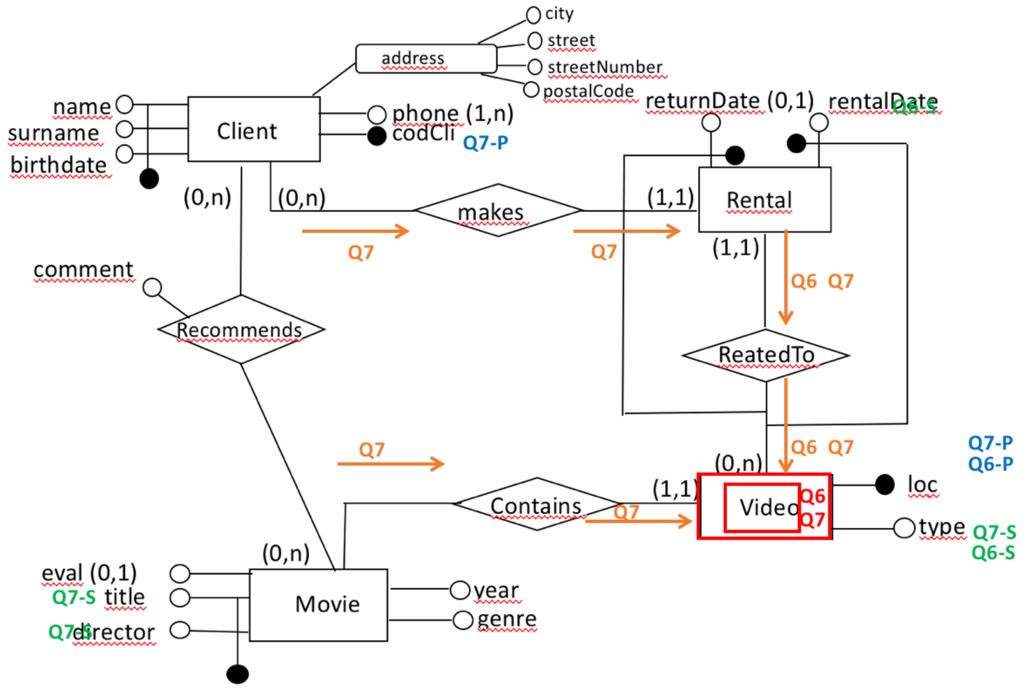
Aggregation entity Video



- Create collection Video
- Query associated with Video
  - Q6, Q7
- Add all the attributes of Video annotated by Q6, Q7
  - {**loc, type,** …
- Now consider Q6
  - One single path of length 1 annotated with Q3: Rental – RelatedTo – Video
  - Cardinality constraint for Video in association RelatedTo: **(0,n)**
  - Add to Video one **set-based attribute** containing tuples composed of attributes of Rental and RelatedTo annotated with Q5
  - video:

    {loc, type, **rentals: [{rental: {rentalDate}}]** , …

Aggregation entity Video

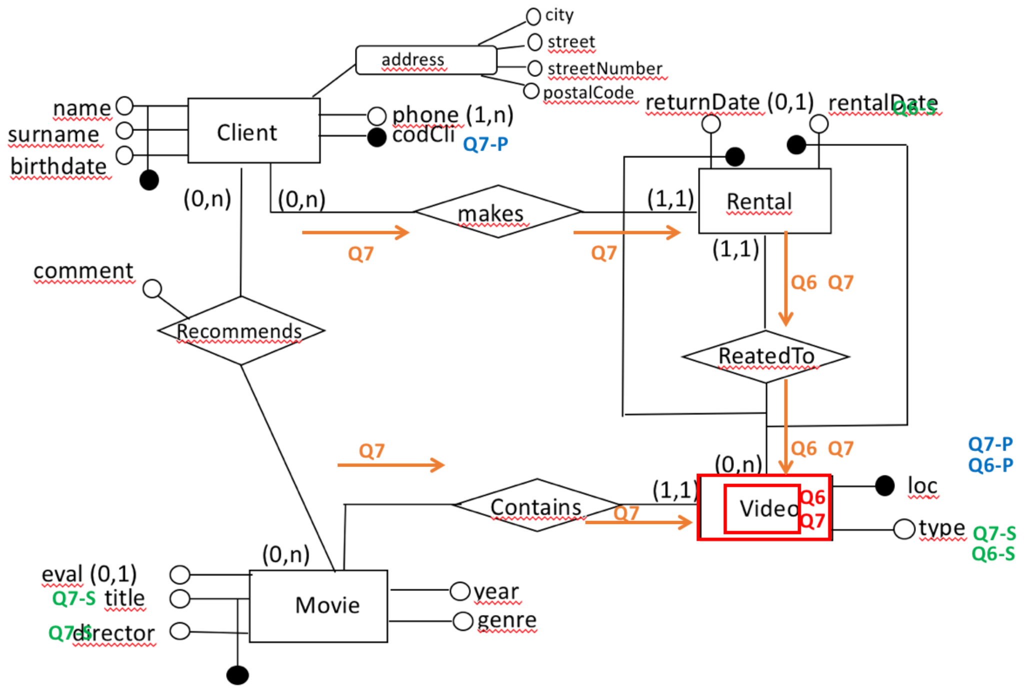


- Now consider Q7, path Client –Makes-Rental-RelatedTo-Video
  - Path of length 2: *Client –Makes-Rental* + *Rental-RelatedTo-Video*
  - Now we analyze each subpath of length 1, starting from Video

- *Rental-RelatedTo-Video: already considered in Q6*
- No attribute in Rental are annotated with Q7 → nothing to do

- *Client-Makes-Rental*: cardinality constraint **(1,1)** from the Rental side
- Add the attributes annotated with Q7 in Client and Makes (only codCli) to the set-based attribute **«rentals» in Video**
- video:{loc, type, rentals: [{rental: {rentalDate, **codcli**}}], …

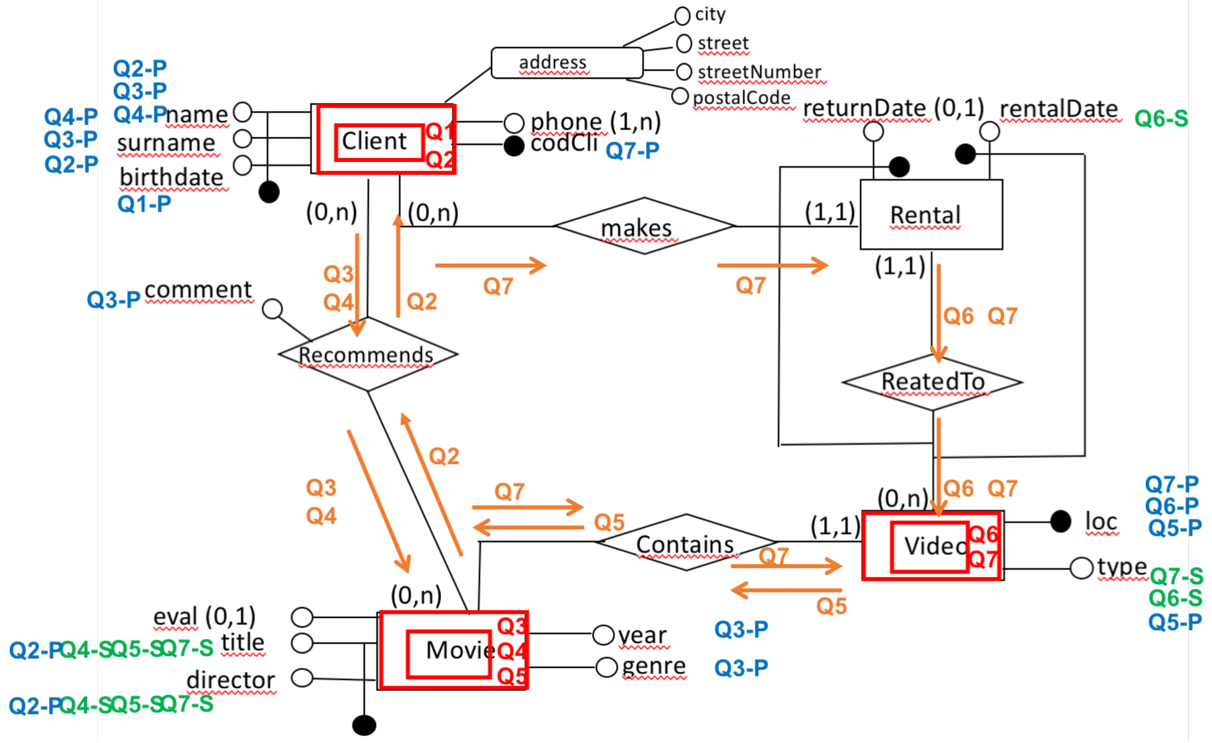
Aggregation entity Video



- Now consider Q7, path Movie-Contains-Video
  - Path with length 1
  - Cardinality constraint for Video in association Contains: **(1,1)**
  - Add to Video the **attributes of** Movie and Contains annotated with Q7
  - video: {loc, type, rentals: [{rental: {rentalDate, codcli}}], **title, director**}

video: {<u>loc</u>, type, rentals: [{rental: {rentalDate, codcli}}], title, director}

Final schema



- client: {name, surname, birthdate, recommends: [{movie: {title, director}}]}
  - Q1, Q2
- movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{video: {loc, type}}]}
  - Q3, Q4, Q5
- video: {loc, type, rentals: [{rental: {rentalDate, codCli}}], title, director}
  - Q6, Q7

Final schema – remarks (1)

- client: {<u>name, surname, birthdate</u>, recommends: [{movie: {title, director}}]}
- movie: {<u>title, director</u>, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{video: {loc, type}}]}
- video: {<u>loc</u>, type, rentals: [{rental: {rentalDate, codCli}}], title, director}

- These elements for items in a collection are optional, they can be added or not

- Possible alternative schema (no movie, video, and rental element, client element added):

- client: {<u>name, surname, birthdate</u>, recommends: [{title, director}]}
- movie: {<u>title, director</u>, year, genre, recommended_by: [{client: {name, surname, comment}}], contained_in: [{loc, type}]}
- video: {<u>loc</u>, type, rentals: [{rentalDate, codCli}], title, director}

An app for aggregate-oriented logical design

- https://amazing-benz-6643f3.netlify.app/

- Example files: https://www.dropbox.com/sh/b64lbe20a0hm1he/AADp_XJfVakMvjqfdUDujDeea?dl=0