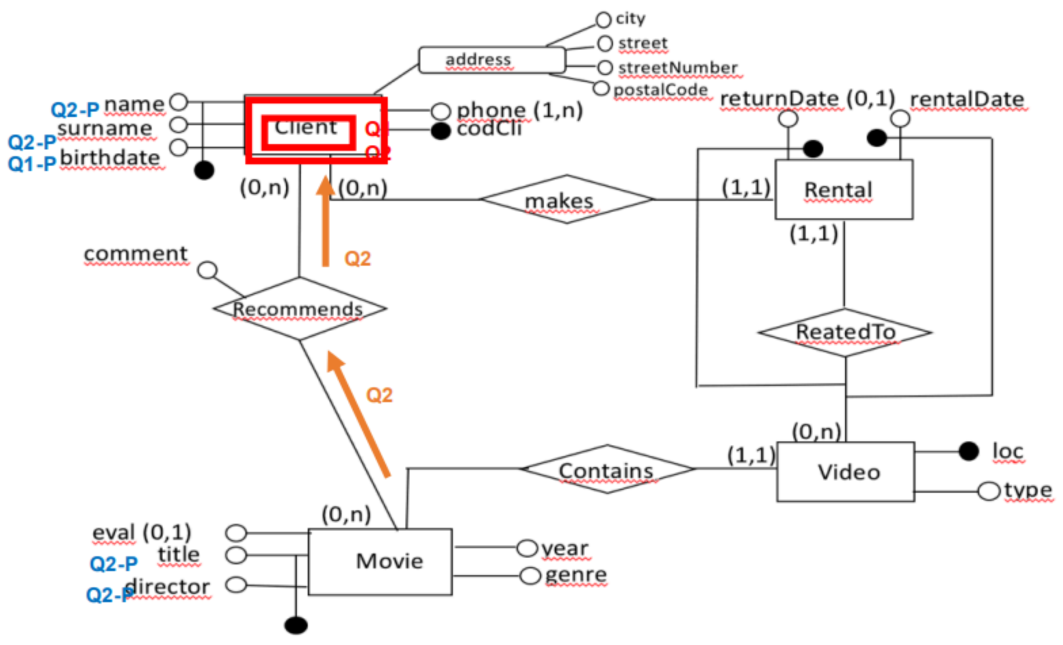
**CASSANDRA flow** 🔥 - *Lorenzo Foschi*



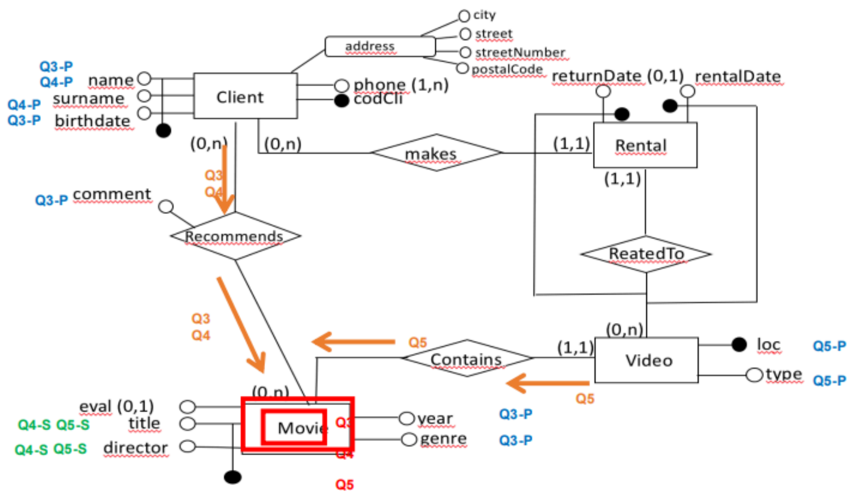
client: {<u>name, surname, birthdate</u>, recommends: [{title, director}]}

**Q1**. Average age of clients
**Q2**. Name and surname of clients and related recommended movies

Selection attributes for Q1: { } & Selection attributes for Q2: { }

**1)** No selection attribute means no need for a specific partition key (i.e we have batch accesses instead of random ones) → So we need to just take the primary key from the ID
**2)** As said before, PK is taken from Client identifier (name, surname, birthday)
**3)** Partition Key = Primary Key = { name, surname, birthday }
Adding something else is fine, but this is the minimal one
**INDEX)** Useless here

```
CREATE TYPE movie_t( title text, director text );
CREATE TABLE Clients (
        name text,
        surname text,
        birthdate date,
        recommends set<frozen<movie_t>>,
        PRIMARY KEY ((name, surname, birthdate))
);
```



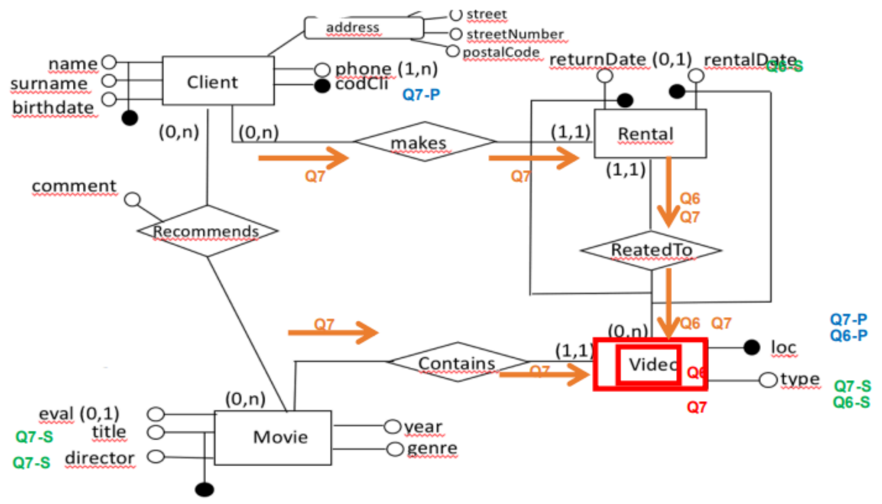
movie: {<u>title, director</u>, year, genre, recommended_by: [{name, surname, comment}], contained_in: [ {loc, type} ]}

**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 'A' by 'B'
**Q5**. Given a movie, all information of videos that contain it

Selection attributes for Q3: { }
Selection attributes for Q4: { title, director } & Selection attributes for Q5: { title, director }

**1)** Both title and director MUST appear in the primary key in order to be able to execute Q4 and Q5. Luckily the intersection between Q4 and Q5 is NOT empty (actually their selection attributes are the exact same, so S(Q4) intersect S(Q5) = S(Q4) = S(Q5)). Also, luckily, the MovieID is also the exact same.
**2)** As said before, PK is taken from Movieidentifier (title, director)
**3)** Partition Key = Primary Key = {title, director}
Also, for Partition Key, title or director alone are ok (but it wouldn't make sense!)
**INDEX)** Useless here

```
CREATE TYPE clientComment_t( name text, surname text, comment text);
CREATE TYPE video_t( loc integer, type text);
CREATE TABLE Movies (
        title text,
        director text,
        year int,
        genre text,
        recommended_by set<frozen<clientComment_t>>,
        contained_in set<frozen<video_t>>,
        PRIMARY KEY ((title, director)
);
```



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

**Q6.** Videos of type 'DVD', rented from a certain date
**Q7.** The videos of type 'VHS' containing the movie 'A' by 'B' and the clients that rented them

Selection attributes for Q6: { rentalDate, type }
Selection attributes for Q7: { title, director, type }

**1)** Type appears in both so it will become the Partition Key (the intersection between Q6 and Q7 is type). Also, we have other three selection attributes:

- title and director, that will be clustering columns
- rentalDate → Problem: it's a frozen type

Unluckily, the MovieID is another one (loc) so it will be added as clustering column
**2)** PK is this sum of selection attributes and ID: { type, title, director, loc }, but rentalDate?
**3)** Partition Key = {type}
**INDEX)** Can be useful to solve the rentalDate problem? NO! Because it's a frozen type and index is not useful here (can be only specified on atomic attributes).

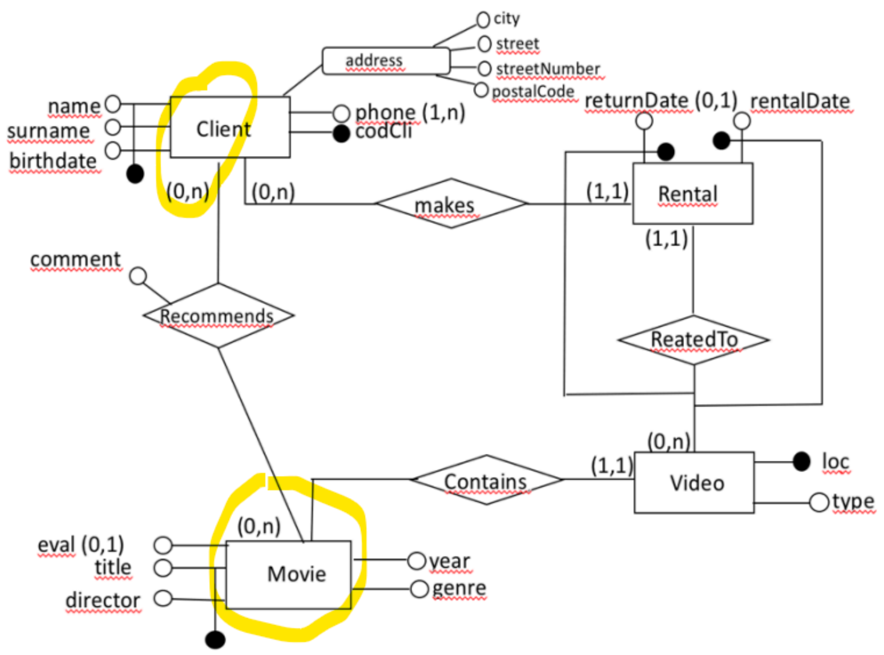
**4)** To solve Q6 let's select, back to aggregation entities, the 1-1 side (Rental):
rental: {rentalDate, loc, type} → the Rental table, together with Videos, solves!

CREATE TYPE clientRentals_t( rentalDate date, codCli int );

CREATE TABLE Videos ( loc int, type text, rentals set<frozen<clientRentals_t>>, title text, director text, PRIMARY KEY (type, title, director, loc));

CREATE TABLE Rentals( rentalDate date, loc int, type text, PRIMARY KEY ((rentalDate, type), loc));

→ Does our solution on point 4) always work? What happens if we have an N-N relationship? We'll see in the next page that in this case we need something more!



**On the query: "***Determine the surname of clients with name «John» that recommended one film produced in 1997"* we can either have:

**Q8-1**(Client, [ Client(name)_!, Movie(year)_R ], [ Client(surname)_! ] )
OR
**Q8-2**(Movie, [ Client(name)_R, Movie(year)_! ], [ Client(surname)_R ] )

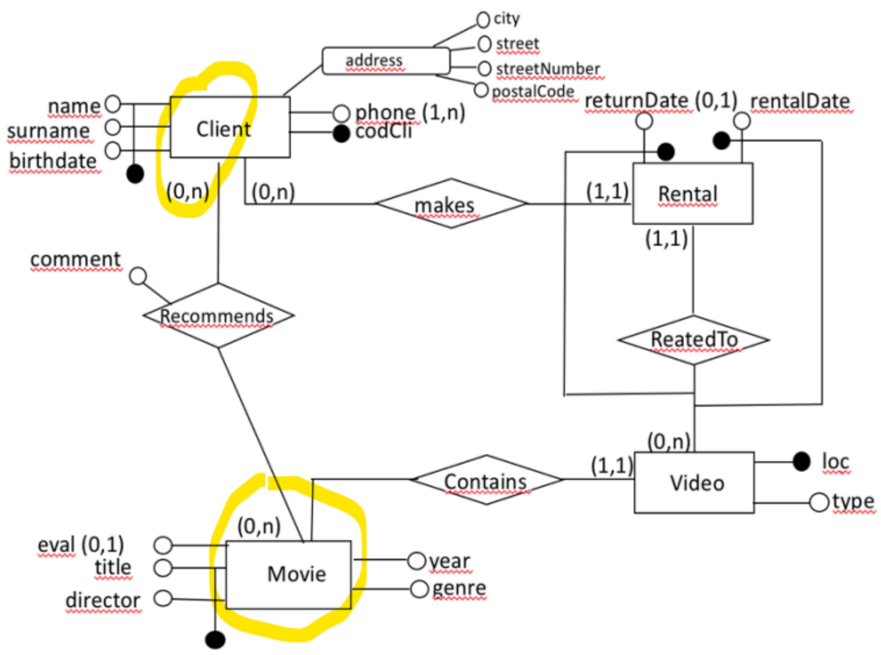
→ On the Client side (**Q8-1**): client: {<u>codcli</u>, name, surname, recommends: [{year}] }

**1)** We have to select both name and year as selection attributes & codcli as ID.
**2)** PK is this sum of selection attributes and ID: { name, codcli }, but how can I specify year, given it's inside a complex type?
**3)** Partition Key = {name}
**INDEX)** Can be useful to solve the year problem? YES! We can CREATE INDEX ON Clients(recommends) and then write the query as follows:

SELECT surname
FROM Clients
WHERE name = «John» AND recommends CONTAINS ''1997''

This is possible because recommends is an array of atomic values so frozen is not needed (that would invalidate the usage of indexes): CREATE TABLE Clients ( codCli int, name text, surname text, recommends set, PRIMARY KEY (name, codCli));

→ On the Movie side (**Q8-2**):
movie: {<u>title, director</u>, year, recommended_by: [{name, surname}] }
Index here can't be used because recommended_by is a set<frozen<map<text, text>>> and therefore the index can't be specified (due to it being a frozen set, so opaque). FAIL, but at least one of the two sides had the index as a solution. Are there even more complex cases?



**With the same schema situation of before, on the query: "***Determine the surname of clients with name «John» that recommended one film produced in 1997, together with the genre of the film*" we can either have:

**Q8-1**(Client, [ Client(name)_!, Movie(year)_R ], [ Client(surname)_!, *Movie(genre)_R* ] )
OR
**Q8-2**(Movie, [ Client(name)_R, Movie(year)_! ], [ Client(surname)_R, *Movie(genre)_!* ] )

→ On the Client side (**Q8-1**): client: {codcli, name, surname, recommends: [{year, genre}] }
FAIL now also even with the index because now we also have genre!
→ On the Movie side (**Q8-2**) obviously FAILS as shown in the previous page!

The solution is to create, as we could do in relational DBs, a new table for the N-N association, named recommendation: {codCli, title, director, name, surname, year, genre}

```
CREATE TABLE Recommendation (
        codCli int,
        name text,
        surname text,
        title text,
        director text,
        year int,
        genre text,
        PRIMARY KEY ((name, year), codCli, title, director
);
```

What if now we add a new query in the workload: “*determine the film produced in 1997*”?

Q9 = ( Movie, [ Movie(year)_! ], [ Movie(title, director)_! ] )

The aggregate doesn’t change with respect to previous workload (see page 2), however…

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

Selection attributes for **Q3**: { }
Selection attributes for **Q4**: { title, director } & Selection attributes for Q5: { title, director }
Selection attributes for **Q9**: { year}

Now S(Q4) intersect S(Q5) intersect S(Q9) is EMPTY (they are disjoint)!
The only solution is to split the aggregate into two column-families:

PRIMARY KEY ((title, director), year) only Q3, Q4, Q5 admitted
PRIMARY KEY (year, title, director) only Q3, Q9 admitted

```
CREATE TABLE Movies (
title text,
director text,
year int,
genre text,
recommended_by set<frozen<map<text, text>>>,
contained_in set<frozen<map<text,text>>>,
PRIMARY KEY ((title, director), year);
```

For queries Q3, Q4, Q5

For queries Q3, Q9

```
CREATE TABLE Movies (
title text,
director text,
year int,
genre text,
recommended_by set<frozen<map<text, text>>>,
contained_in set<frozen<map<text,text>>>,
PRIMARY KEY (year, title, director);
```

Or, given that Q3 is already supported by Movies(1), Movies(2) can be just:

For queries Q9

```
CREATE TABLE Movies (
title text,
director text,
year int,
PRIMARY KEY (year, title, director);
```

*The summary of this is: thanks Cassandra for being part of my life…* ✨