

COMP2400

Relational Databases

Lecture 31: Query Planning

Ben Lippmeier
Australian National University
Semester 2
2008

Movie database example

movie	
code	: varchar(16)
title	: varchar(80)
year	: int
director	: varchar(80)
producer	: varchar(80)
studio	: varchar(40)
process	: varchar(40)
category	: varchar(40)
awards	: varchar(80)
locale	: varchar(20)
notes	: text

11470 rows

casts	
code	: varchar(16)
actor_name	: varchar(40)
actor_role	: text

44499 rows

actor	
stage_name	: varchar(80)
given_name	: varchar(40)
last_name	: varchar(40)
sex	: char(1)
birth_year	: int
death_year	: int
role	: varchar(80)
origin	: varchar(40)
notes	: text

6805 rows

- Derived from data by Gio Wiederhold, Stanford University.
- Source: UCI KKD Archive. <http://kkd.ics.uci.edu>

Query Planning

```
SELECT title, year, stage_name, birth_year
FROM    movie, casts, actor
WHERE   movie.code = casts.code
          AND stage_name = actor_name
          AND year - birth_year <= 40
          AND sex      = 'F'
          AND director = 'Fellini'
```

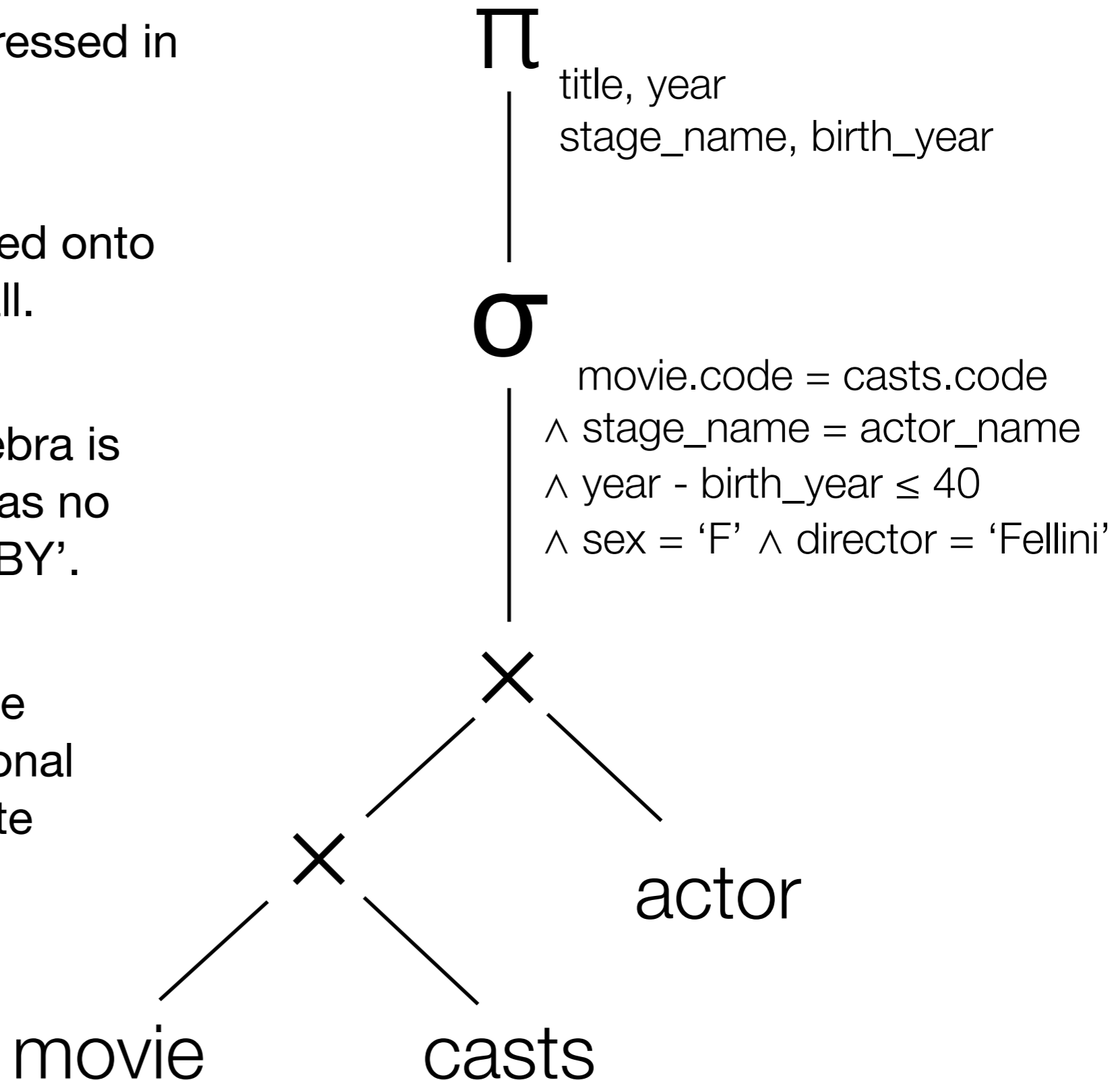
- We will use this example for the rest of the lecture
- We would like to know which actresses featured in Fellini's films who were under 40 at the time of their appearance.

- This is the same query expressed in relational algebra.

- Much of SQL can be mapped onto relational algebra, but not all.

- For example, relational algebra is based on sets and hence has no direct support for 'ORDER BY'.

- Real DBMS systems will use varying extensions of relational algebra for their intermediate languages, but the general ideas are the same.



Reminder about Cross Join / Cartesian Product

- The cartesian product of two sets A and B is written $A \times B$.
- It is defined in terms of a set comprehension:

$$A \times B = \{ (a, b) \mid a \in A, b \in B \}$$

- Cartesian product is defined on *sets*,
so the resulting rows / tuples could come out in any order.

Example Cross Join

fid	name
12	apple
10	banana
5	cherry
27	durian

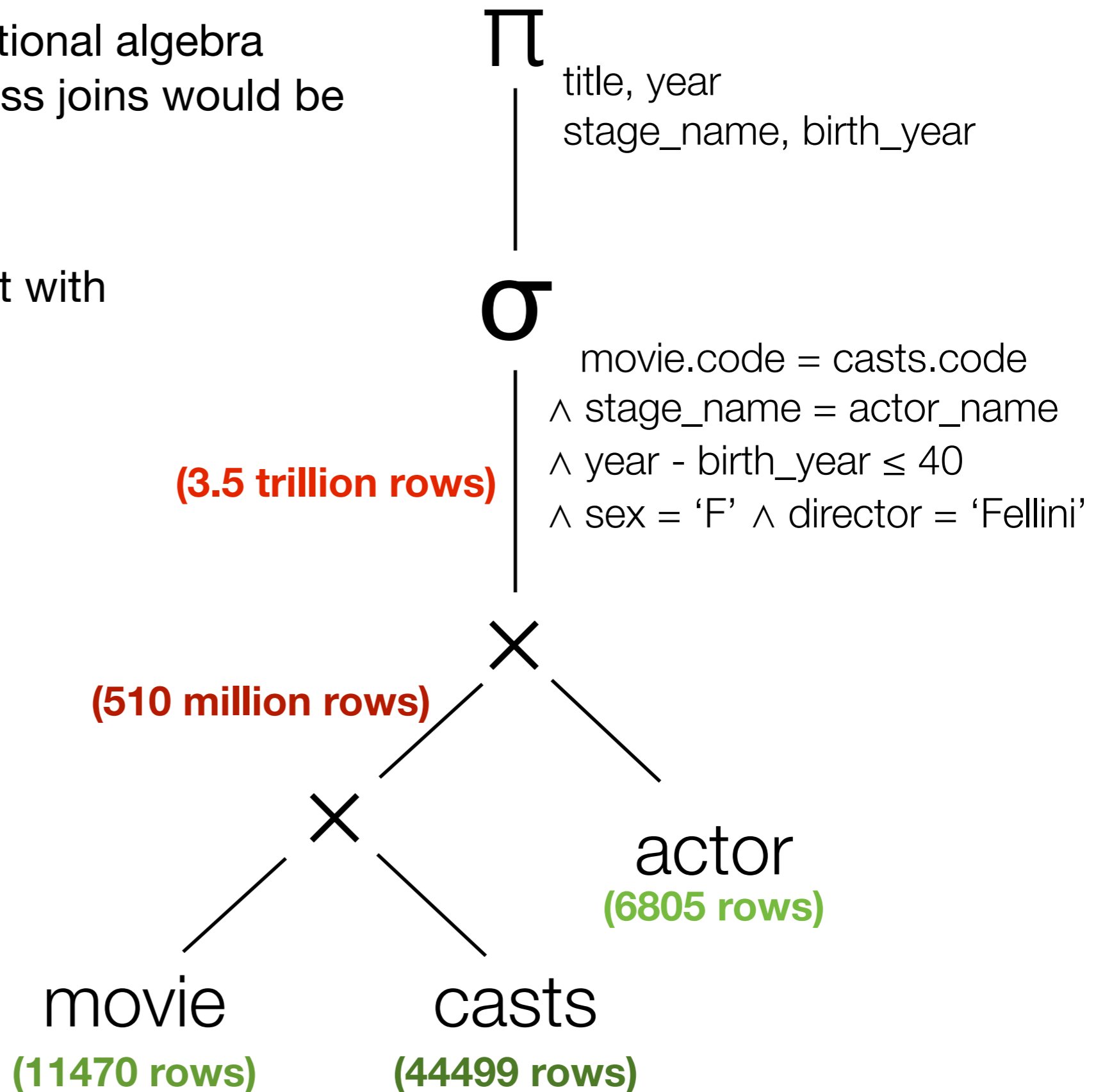
×

lid	price
5	0.25
10	1.10
12	0.80
15	15.50

=

fid	name	lid	price
12	apple	5	0.25
12	apple	10	1.10
12	apple	12	0.80
12	apple	15	15.50
10	banana	5	0.25
10	banana	10	1.10
10	banana	12	0.80
10	banana	15	15.50
5	cherry	5	0.25
5	cherry	10	1.10
5	cherry	12	0.80
5	cherry	15	15.50
27	durian	5	0.25
27	durian	10	1.10
27	durian	12	0.80
27	durian	15	15.50

- Trying to evaluating a relational algebra expression containing cross joins would be a disaster!
- The result of $R \times S$ is a set with $|R| * |S|$ elements.
- We need to compute this result in a more feasible manner.



Heuristic rewrites

- We will make use of algebraic identities to rewrite our expression / operator tree into a different form.
- This new form will express the same value, but it will be possible to implement it more efficiently on a physical machine.
- We will also have to map our relational operators onto physical operators. We should keep the physical operators in mind when performing rewrites.
- Some of our rewrites will *always* result in a faster query.
- Some will *sometimes* help, and they won't hurt performance otherwise.
- Others will sometimes help, and sometimes make matters *worse*. Worrying about these is the job of the query optimiser.

Some Relational Algebra Identities

- Selection

$$\sigma_A(\sigma_A(R)) \equiv \sigma_A(R) \quad \text{idempotence}$$

$$\sigma_A(\sigma_B(R)) \equiv \sigma_B(\sigma_A(R)) \quad \text{commutativity}$$

$$\sigma_A(\sigma_B(R)) \equiv \sigma_{A \wedge B}(R) \quad \text{conjunction}$$

- Selection before product.

$$\sigma_{ABC}(R \times S) \equiv \sigma_A(\sigma_B(R) \times \sigma_C(S))$$

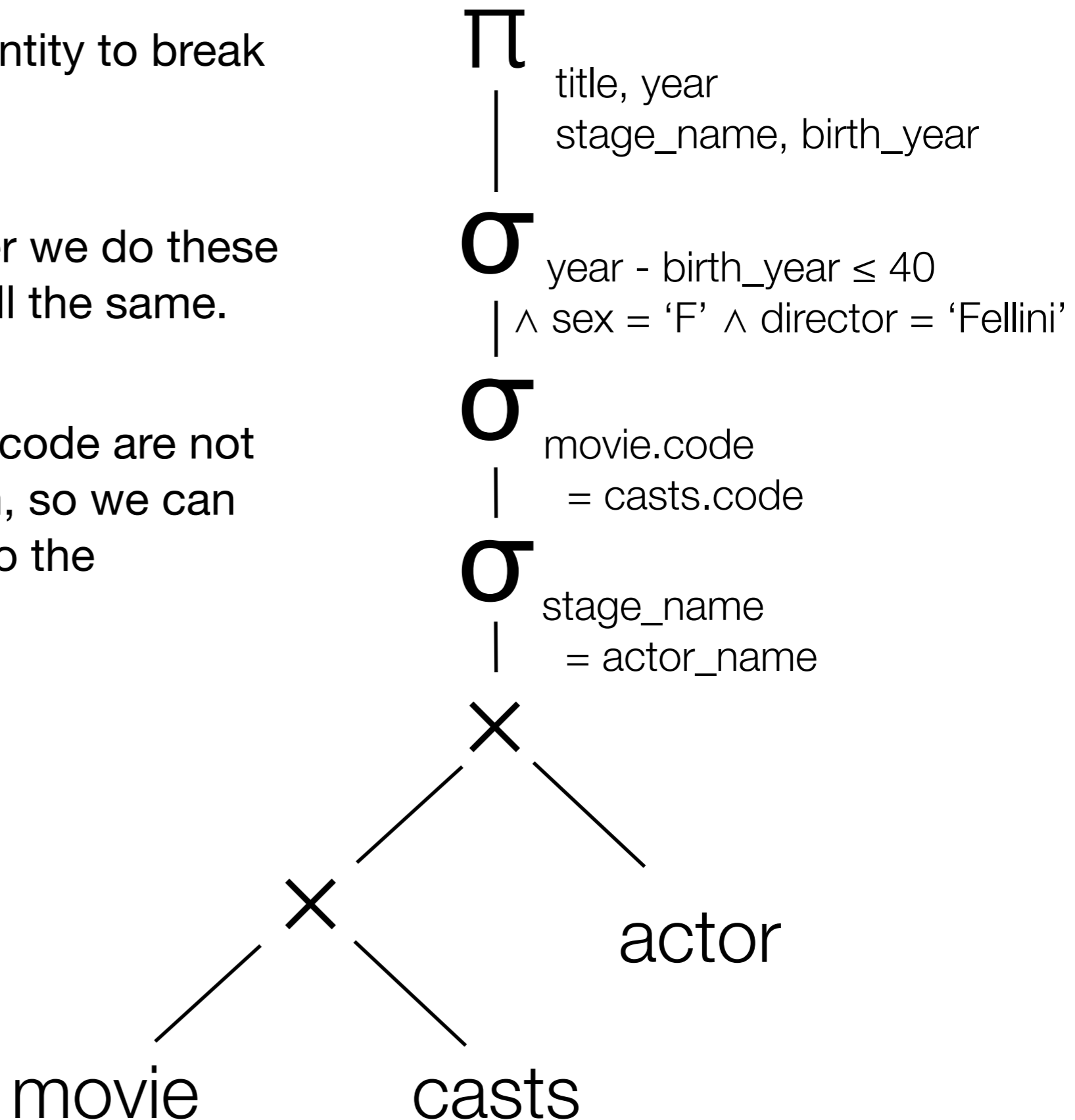
where the selection condition

B only contains attributes in R and

C only contains attributes in S .

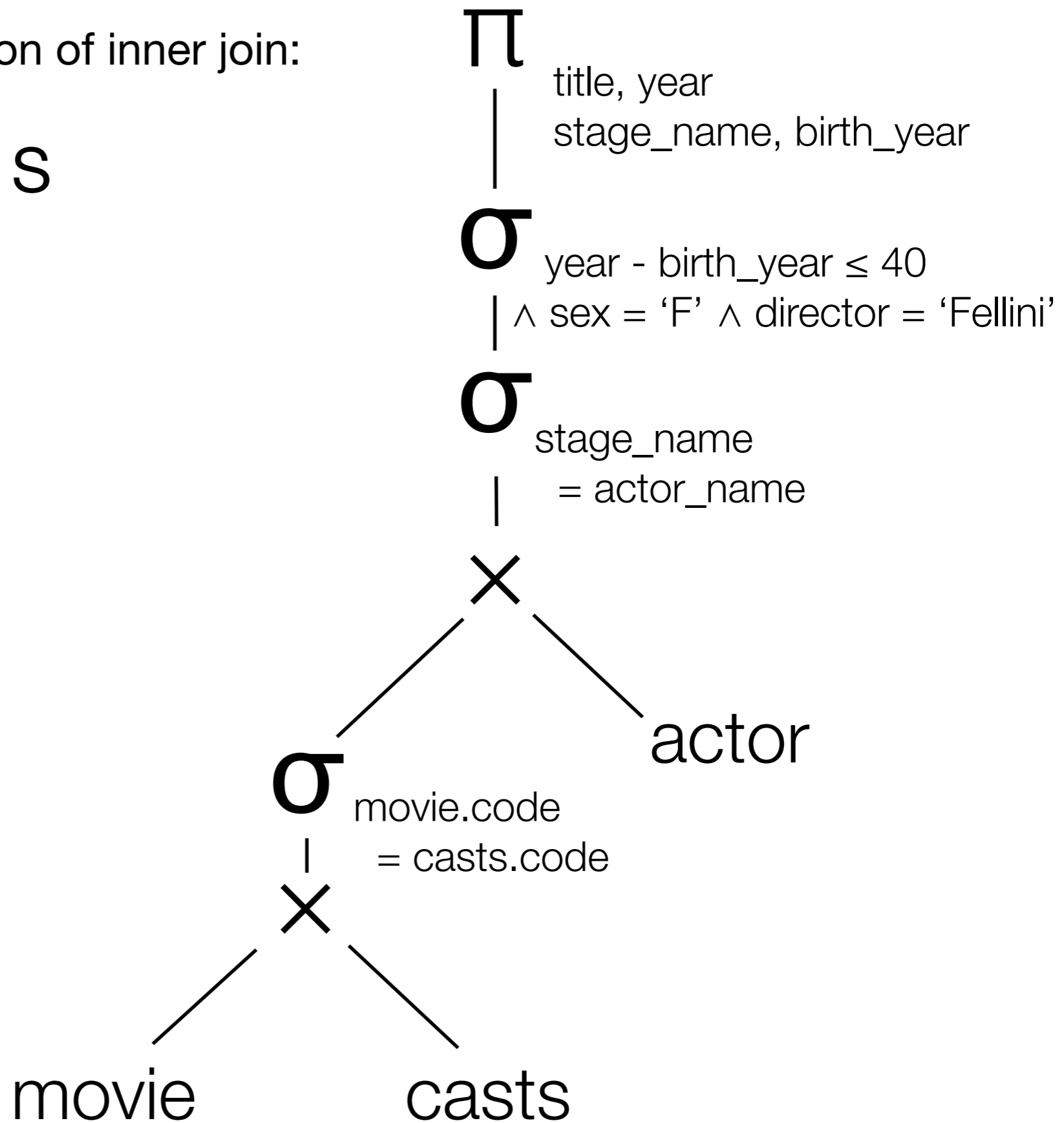
A can contain comparisons between attributes in both R and S .

- First use the conjunction identity to break up the selection operator.
- It does not matter what order we do these selections in, the result is still the same.
- Also, movie.code and casts.code are not present in the actors relation, so we can push this selection down into the left branch.

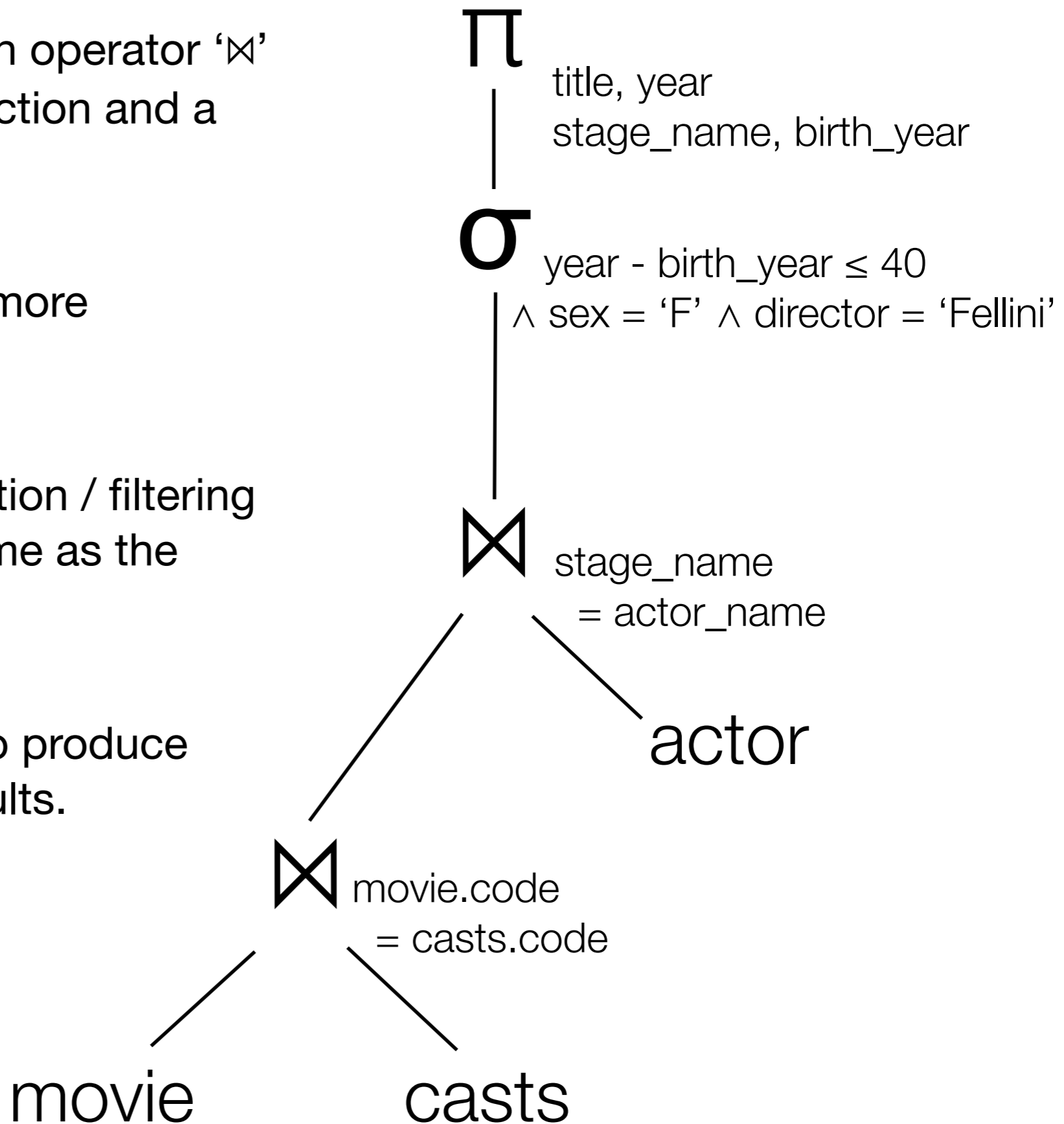


- Now we can use the definition of inner join:

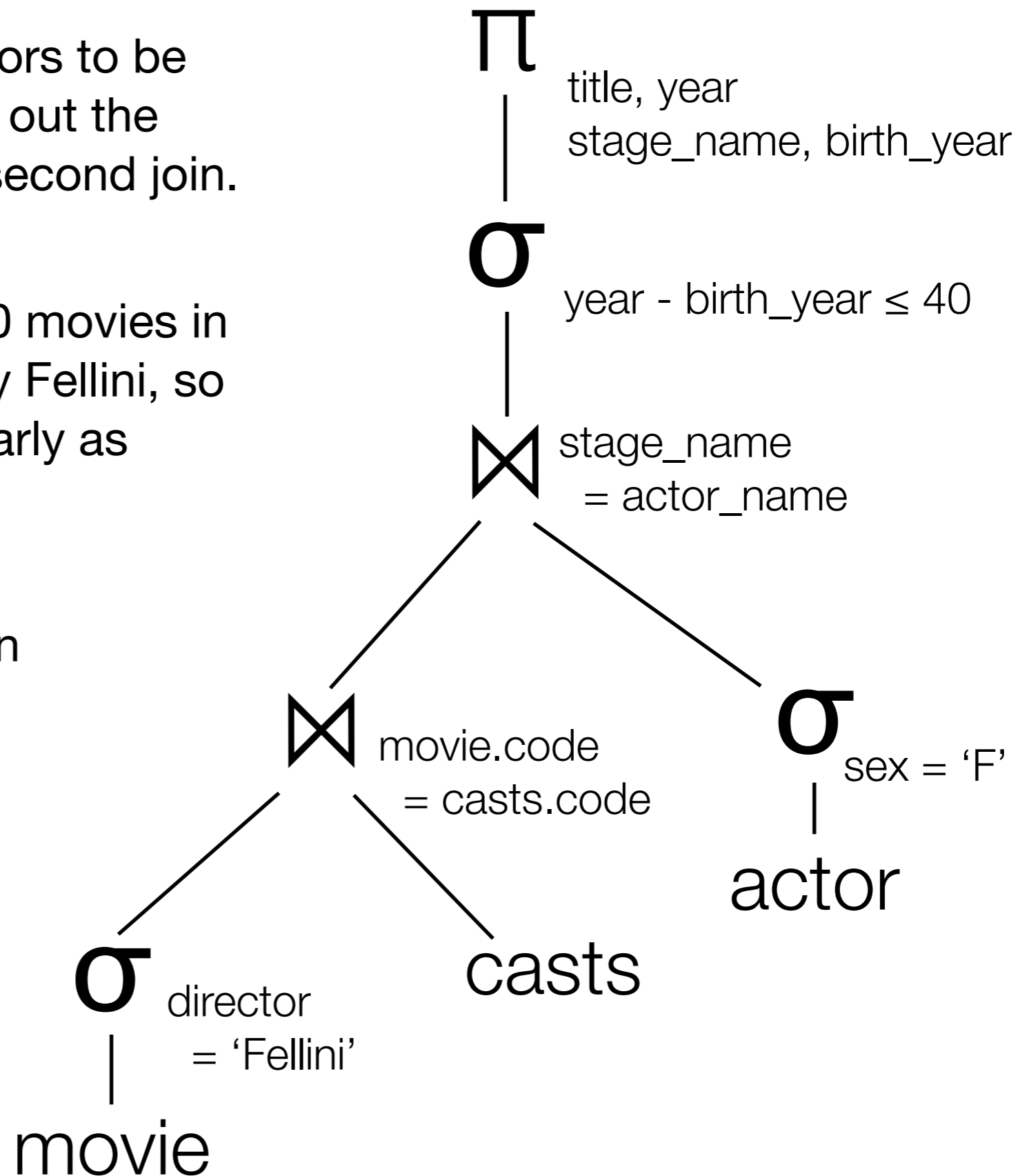
$$\sigma_A(R \times S) \equiv R \bowtie_A S$$



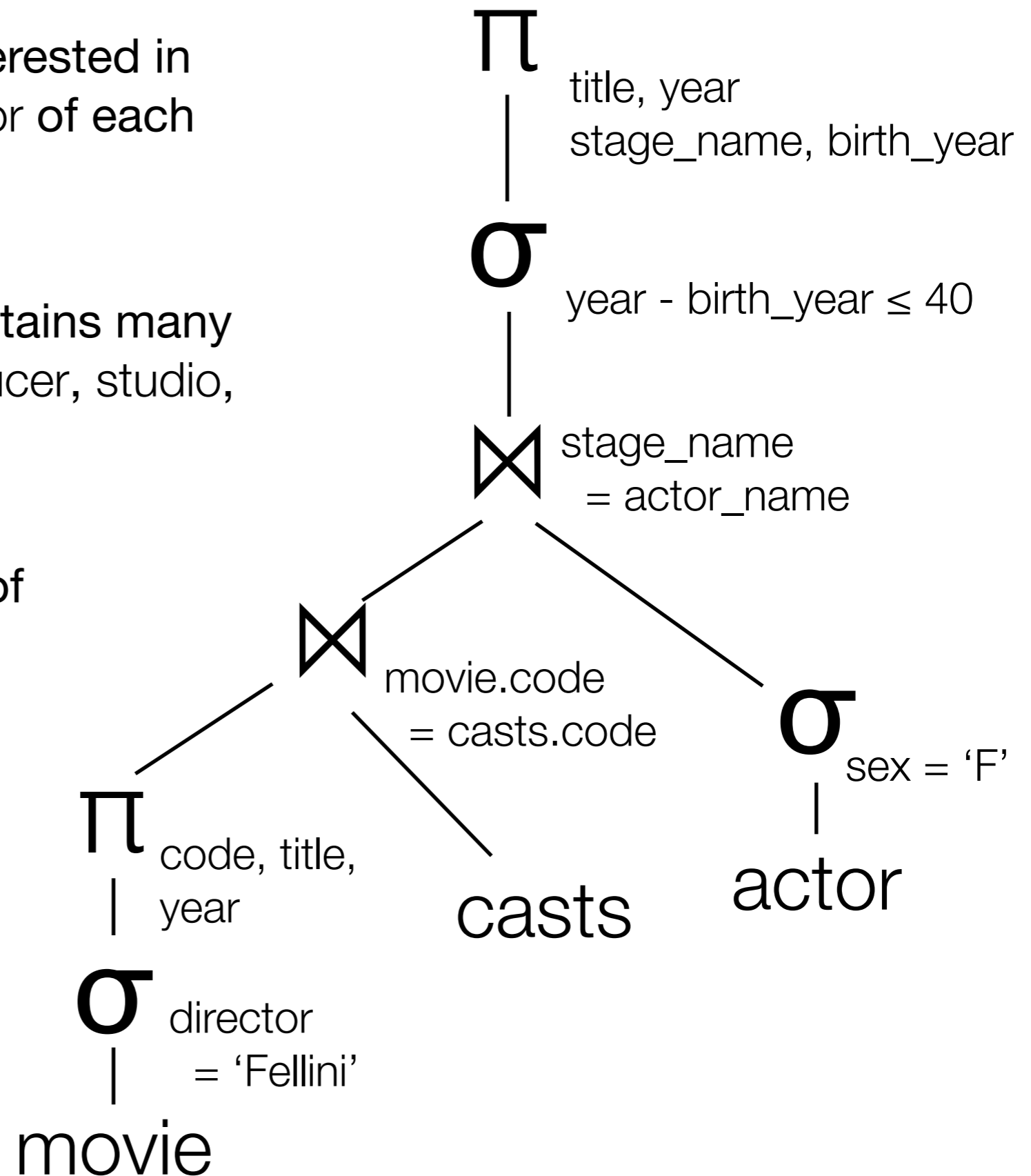
- We can think of the inner join operator ‘ \bowtie ’ as *syntactic sugar* for a selection and a cartesian product.
- But the idea of “join” has a more operational meaning.
- We now intend for the selection / filtering to take place at the same time as the joining / linking of tuples.
- This saves us from having to produce streams of intermediate results.



- We only expect half of the actors to be female, so it's better to filter out the males *before* performing the second join.
- Likewise, only 19 out of 11470 movies in the database were directed by Fellini, so it's better to select these as early as possible.
- We can't push the selection on $(year - birth_year \leq 40)$ deeper into the tree because it uses attributes from both branches of the outer most join.



- For this query we are only interested in the title, year, code and director of each movie.
- However, our movie table contains many other attributes such as producer, studio, process and awards.
- If we have to store the result of joining movie and casts back to disk, then we can save space by discarding non-interesting attributes early on.



Some more identities

- Here are some other identities which may be useful in a different query. Can you think of any others?

- Projection

$$\pi_{as} (\pi_{bs}(R)) \equiv \pi_{as} \quad \text{where } as \subseteq bs$$

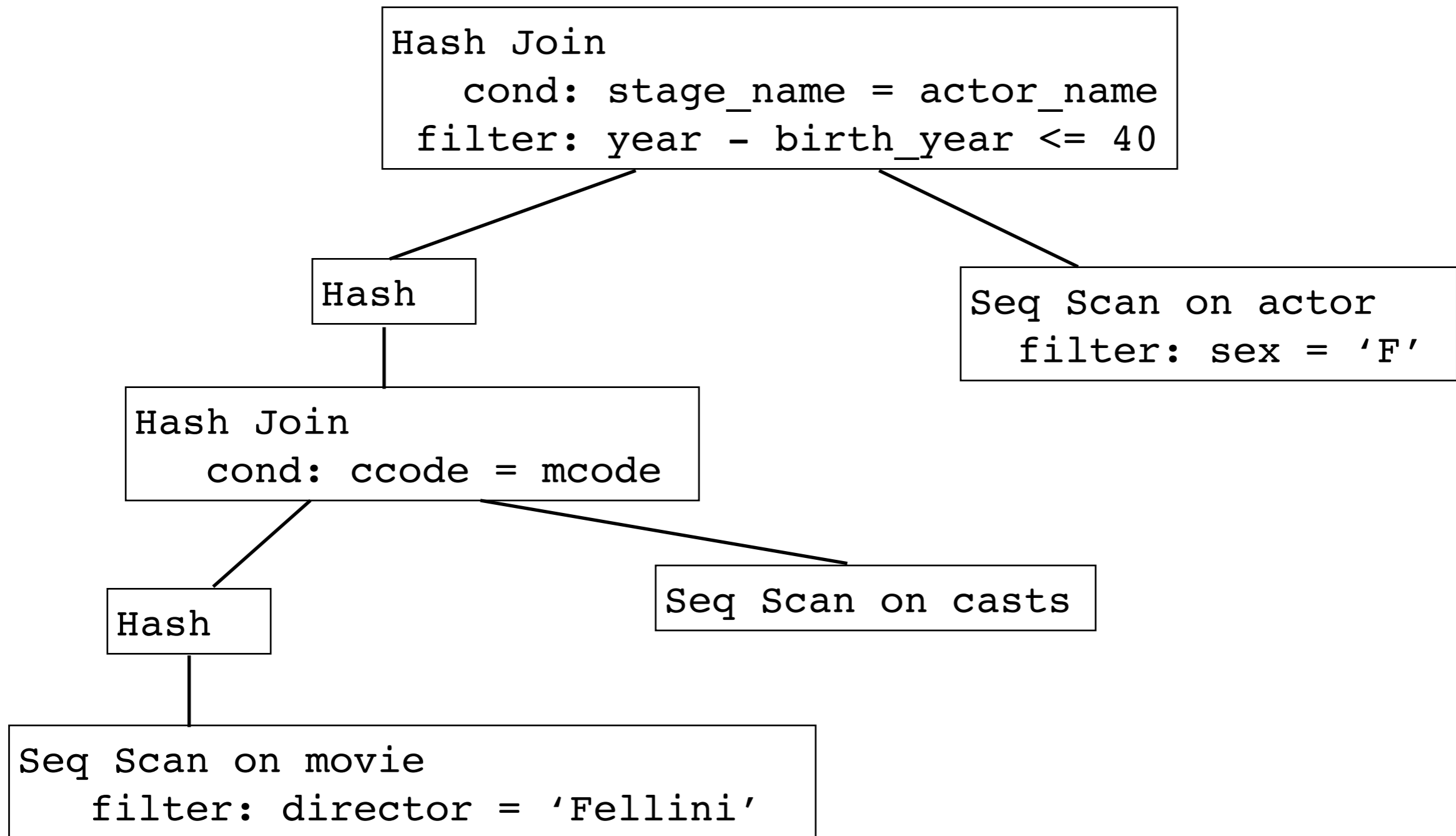
- Selection with Projection

$$\sigma_A (\pi_{as}(R)) \equiv \pi_{as}(\sigma_A(R))$$

- Cartesian Product.

$$R \times S \equiv S \times R \quad \text{commutativity}$$

$$(R \times S) \times T \equiv R \times (S \times T) \quad \text{associativity}$$



- This is PostgreSQL's plan, obtained with EXPLAIN ANALYSE.

QUERY PLAN

Hash Join

(cost=1740.98..1904.69 rows=3 width=37)

(actual time=56.826..64.197 rows=7 loops=1)

Hash Cond: ((actor.stage_name)::text = (casts.actor_name)::text)

Join Filter: ((movie.year - actor.birth_year) <= 40)

-> Seq Scan on actor

(cost=0.00..154.06 rows=2536 width=17)

(actual time=0.038..5.682 rows=2550 loops=1)

Filter: ((sex)::text = 'F'::text)

-> Hash

(cost=1740.68..1740.68 rows=24 width=33)

(actual time=56.702..56.702 rows=82 loops=1)

-> Hash Join

(cost=504.46..1740.68 rows=24 width=33)

(actual time=23.167..**56.536** rows=82 loops=1)

Hash Cond: ((casts.code)::text = (movie.code)::text)

-> Seq Scan on casts

(cost=0.00..790.99 rows=44499 width=18)

(actual time=0.018..16.032 rows=44499 loops=1)

-> Hash

(cost=504.39..504.39 rows=6 width=25)

(actual time=11.244..11.244 rows=19 loops=1)

-> Seq Scan on movie

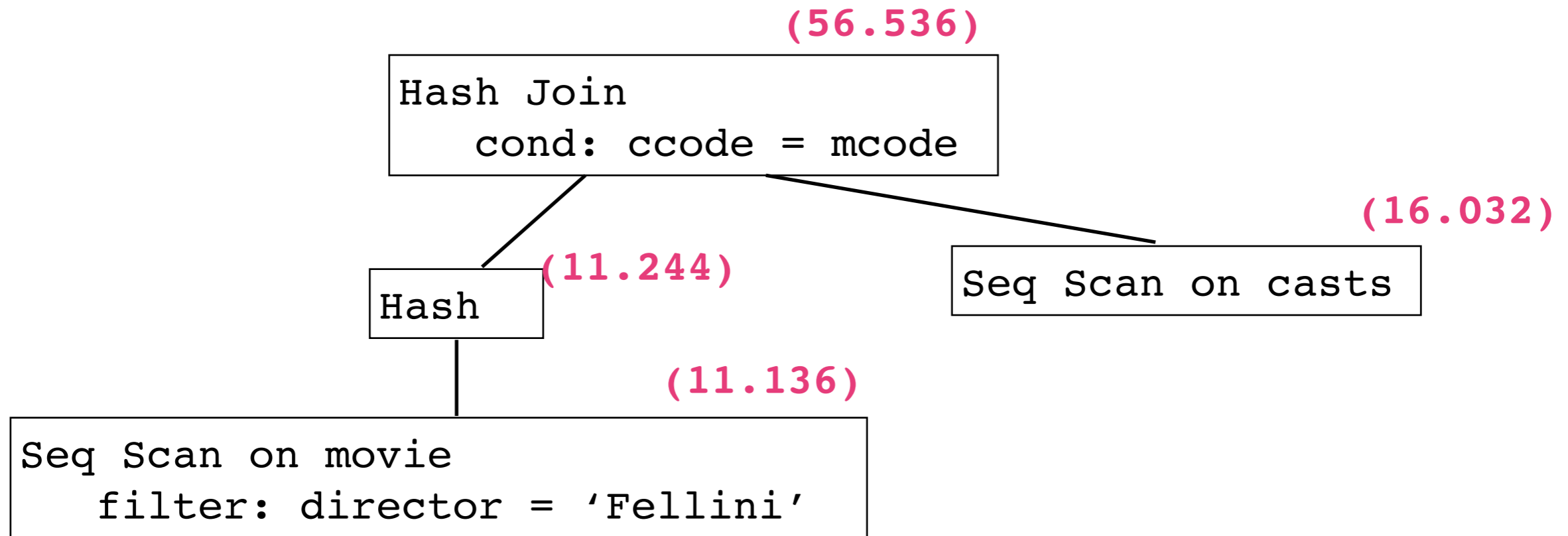
(cost=0.00..504.39 rows=6 width=25)

(actual time=5.435..11.163 rows=19 loops=1)

Filter: (director = 'Fellini'::text)

Total runtime: **64.573** ms

Where does the time go?



- Times quoted are milliseconds consumed by that operator and *all its children*.
- Most of our time is being spent matching up cast members with the films they featured in.
- Only 19 rows are returned by the left branch, but there are 44499 rows in the casts table. Sequentially scanning it is probably a bad idea...

Create an index for casts

```
CREATE INDEX casts_code ON casts (code);
```

↑
index name

↑ ↑
table expression

- By default, PostgreSQL creates a B+Tree clustered index.
- You can create an index on any expression involving attributes from the table, but (code) suffices for our purposes.
- Now the DBMS should be able to find all the cast members that featured in a particular film, just by consulting the casts_code index.

QUERY PLAN

Hash Join

(cost=578.15..741.86 rows=3 width=37)

(actual time=11.147..18.553 rows=7 loops=1)

Hash Cond: ((actor.stage_name)::text = (casts.actor_name)::text)

Join Filter: ((movie.year - actor.birth_year) <= 40)

-> Seq Scan on actor

(cost=0.00..154.06 rows=2536 width=17)

(actual time=0.026..5.740 rows=2550 loops=1)

Filter: ((sex)::text = 'F'::text)

-> Hash

(cost=577.85..577.85 rows=24 width=33)

(actual time=11.035..11.035 rows=82 loops=1)

-> Nested Loop

(cost=0.00..577.85 rows=24 width=33)

(actual time=3.890..**10.852** rows=82 loops=1)

-> Seq Scan on movie

(cost=0.00..504.39 rows=6 width=25)

(actual time=3.813..9.508 rows=19 loops=1)

Filter: (director = 'Fellini'::text)

-> **Index Scan using casts_code on casts**

(cost=0.00..12.16 rows=7 width=18)

(actual time=0.061..0.066 rows=4 loops=19)

Index Cond: ((casts.code)::text = (movie.code)::text)

Total runtime: **18.811** ms

Only half the actors are female...

```
CREATE INDEX actor_sex ON actor (sex);
```

QUERY PLAN

```
Hash Join (cost=578.15..741.86 rows=3 width=37)
  Hash Cond: ((actor.stage_name)::text = (casts.actor_name)::text)
  Join Filter: ((movie.year - actor.birth_year) <= 40)
  -> Seq Scan on actor (cost=0.00..154.06 rows=2536 width=17)
      Filter: ((sex)::text = 'F'::text)
  -> Hash (cost=577.85..577.85 rows=24 width=33)
      -> Nested Loop (cost=0.00..577.85 rows=24 width=33)
          -> Seq Scan on movie (cost=0.00..504.39 rows=6 width=25)
              Filter: (director = 'Fellini'::text)
          -> Index Scan using casts_code on casts
              (cost=0.00..12.16 rows=7 width=18)
              Index Cond: ((casts.code)::text = (movie.code)::text)
```

- But in this case the query planner has decided that it will be quicker just to do a sequential scan.
- Indexes tend help more when the query has a high *selectivity*.

Not all Roses

- Index take time to create, and on some DBMS systems all other access to the table is blocked while it is being created.

This can be unacceptable for live systems.

- Indexes slow down update, insert and delete operations as we must also maintain index entries for each row.
- Indexes take up space on the disk, and consume part of our cache.
- Indexes which are not being used should be removed from the database.

Partial Indexes

```
CREATE INDEX movie_director_fellini  
ON      movie (director)  
WHERE director = 'Fellini'
```

- If our queries tend to refer to certain values over others then we can create a partial index.
- A partial index only has an entries for a subset of the rows in a table.
- Partial indexes can take up significantly less space on the disk.
- We'll see how to check how much space they're using next lecture.

QUERY PLAN

Hash Join

(cost=99.81..263.52 rows=3 width=37)

(actual time=1.720..10.325 rows=7 loops=1)

Hash Cond: ((actor.stage_name)::text = (casts.actor_name)::text)

Join Filter: ((movie.year - actor.birth_year) <= 40)

-> Seq Scan on actor

(cost=0.00..154.06 rows=2536 width=17)

(actual time=0.033..6.576 rows=2550 loops=1)

Filter: ((sex)::text = 'F'::text)

-> Hash

(cost=99.51..99.51 rows=24 width=33)

(actual time=1.608..1.608 rows=82 loops=1)

-> Nested Loop

(cost=4.30..99.51 rows=24 width=33)

(actual time=0.135..1.431 rows=82 loops=1)

-> **Bitmap Heap Scan on movie**

(cost=4.30..26.05 rows=6 width=25)

(actual time=0.065..0.079 rows=19 loops=1)

Recheck Cond: (director = 'Fellini'::text)

-> **Bitmap Index Scan on movie_director_fellini**

(cost=0.00..4.30 rows=6 width=0)

(actual time=0.054..0.054 rows=19 loops=1)

Index Cond: (director = 'Fellini'::text)

-> Index Scan using casts_code on casts

(cost=0.00..12.16 rows=7 width=18)

(actual time=0.059..0.063 rows=4 loops=19)

Index Cond: ((casts.code)::text = (movie.code)::text)

Total runtime: **10.596** ms

Garbage In, Garbage Out

- When collecting DBMS performance measurements, do not forget the effect of caching in the HDD firmware, HDD controller and OS kernel.
- If a query takes $< 10\text{ms}$ it's probably not accessing the disk at all.
- A modern OS will aggressively cache data read from disk, and the cache can grow to fill all available memory.
- If all your data is in cache then your queries will run blindingly fast, but this might not be the case a few minutes later..



To flush the caches on UNIX-a-likes

- Shut down the DBMS. You will need to be the super-user to do this..

```
sudo -u postgres -D <data path> -m immediate stop
```

- Create a dummy file the size of physical memory.

We can just fill it full of zeros.

Ensure this file is really on disk. Some OSs treat /tmp specially.

```
dd if=/dev/zero of=/tmp/dummy bs=1024 count=1000000
```

- Force the dummy file to be read off disk (and cached)

The data can be discarded once it is read.

```
cat /tmp/dummy > /dev/null
```

- Restart the DBMS.

```
sudo -u postgres -D <data path> start
```

A grain of salt.

- ... but what part of that flushed cache held the DBMS binary vs table data?
- Perhaps by flushing the cache so aggressively, the first query the DBMS executes will now take much longer than it otherwise would...
- Be very wary of extrapolating data-base performance tests to other situations.
- It's much better to test a system under a real load.

If you're worried about breaking the live database, then capture a stream of transactions and play it back to a mirrored version.

- Experiment, add indexes, check the output of EXPLAIN ANALYSE, and refine.