L20: Joins

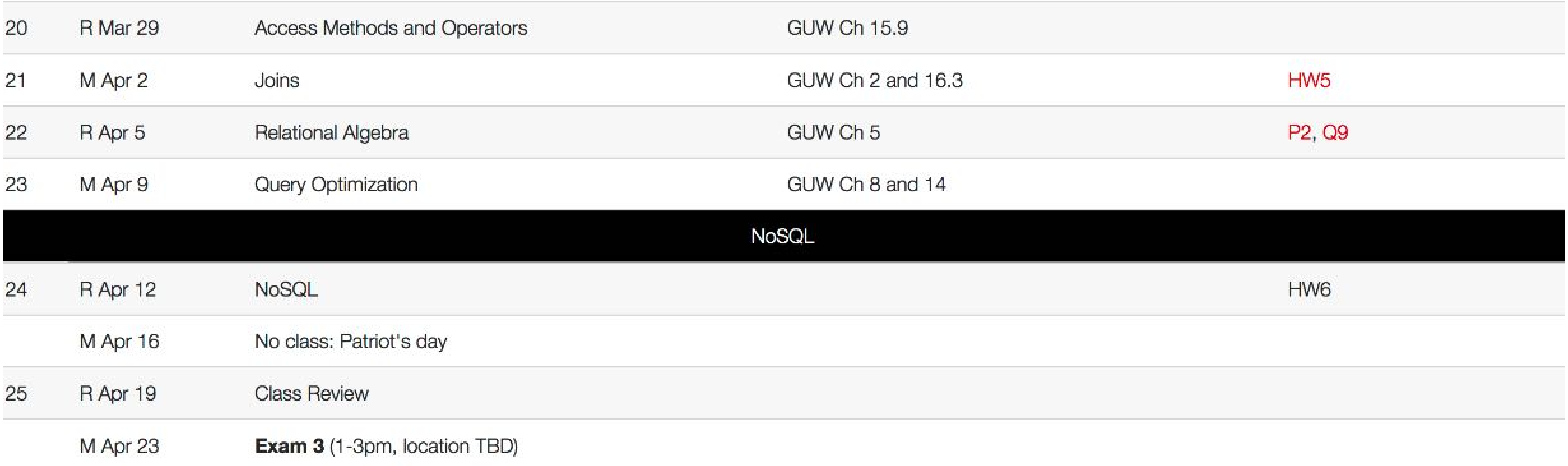
CS3200 Database design (sp18 s2)

https://course.ccs.neu.edu/cs3200sp18s2/

3/29/2018

Announcements!

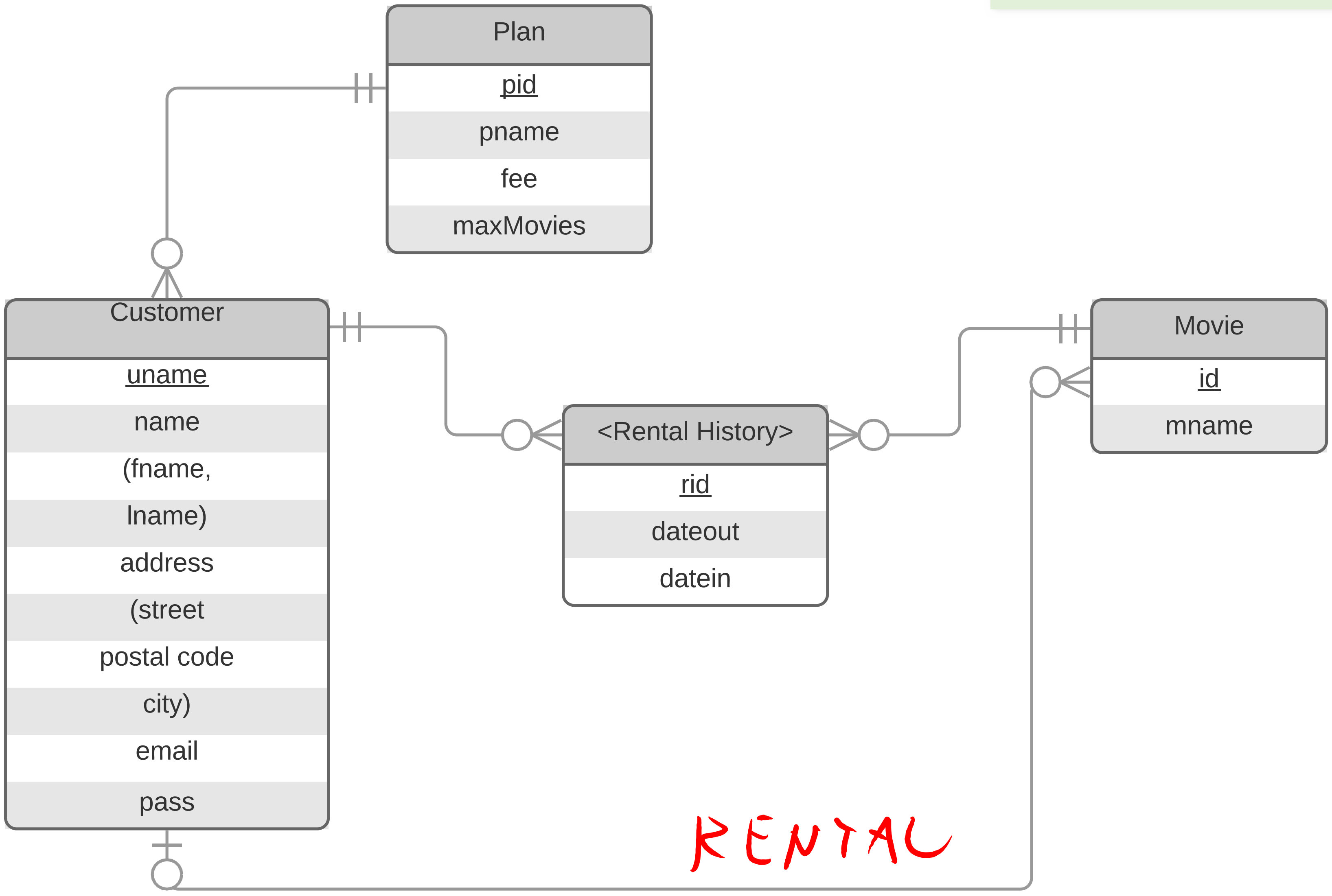
- Please pick up your exam if you have not yet (at end of class)
- We leave some space at the end of today to find your project mates
- Hands-on experience with NoSQL?
- Feedback: calculations were difficult last time, we will go slower and repeat
- Outline today
  - Fun with indexing in Postgresql
  - Joins



20	R Mar 29	Access Methods and Operators	GUW Ch 15.9	
21	M Apr 2	Joins	GUW Ch 2 and 16.3	HW5
22	R Apr 5	Relational Algebra	GUW Ch 5	P2, Q9
23	M Apr 9	Query Optimization	GUW Ch 8 and 14	
		NoSQL		
24	R Apr 12	NoSQL		HW6
	M Apr 16	No class: Patriot's day		
25	R Apr 19	Class Review		
	M Apr 23	**Exam 3** (1-3pm, location TBD)		

Project phase 1: example solution

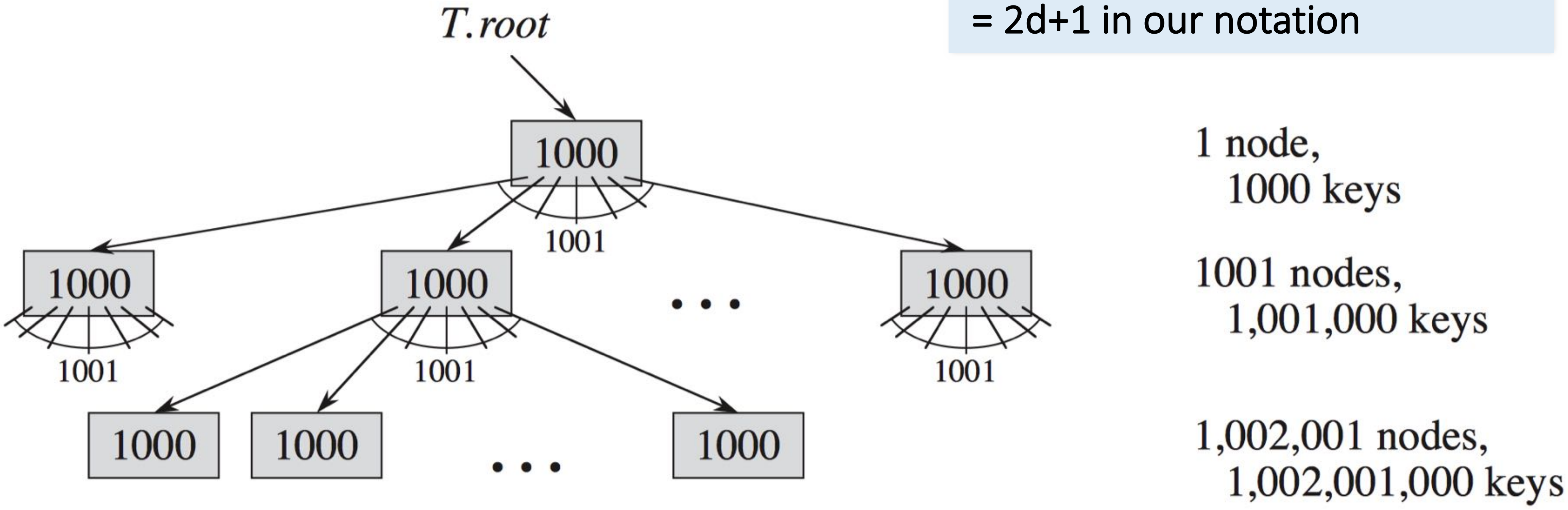
the content provider will only allow you to rent each movie to at most one customer at any one time.



Notice different B-tree notations

We define the degree as the minimum number of keys.
Notice that CLRS defines it as minimum number of children.
MIN # children = MIN # of keys + 1

Order… max number of children
= 2d+1 in our notation



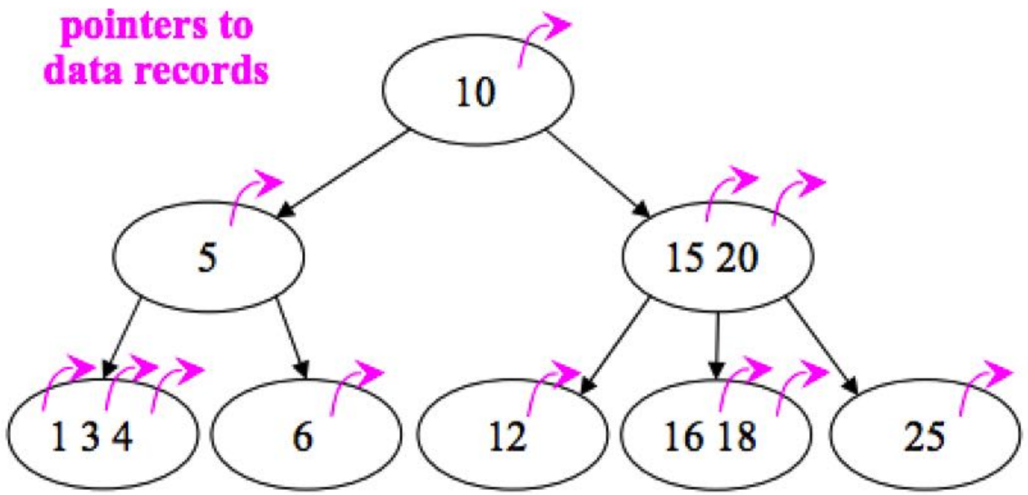
Source: CLRS Algorithms: 2009 - Cormen, Leiserson, Rivest, Stein - Introduction to algorithms (book, 3rd ed)

B-tree: (min) degree d (min # of keys). Of "order" 2d+1

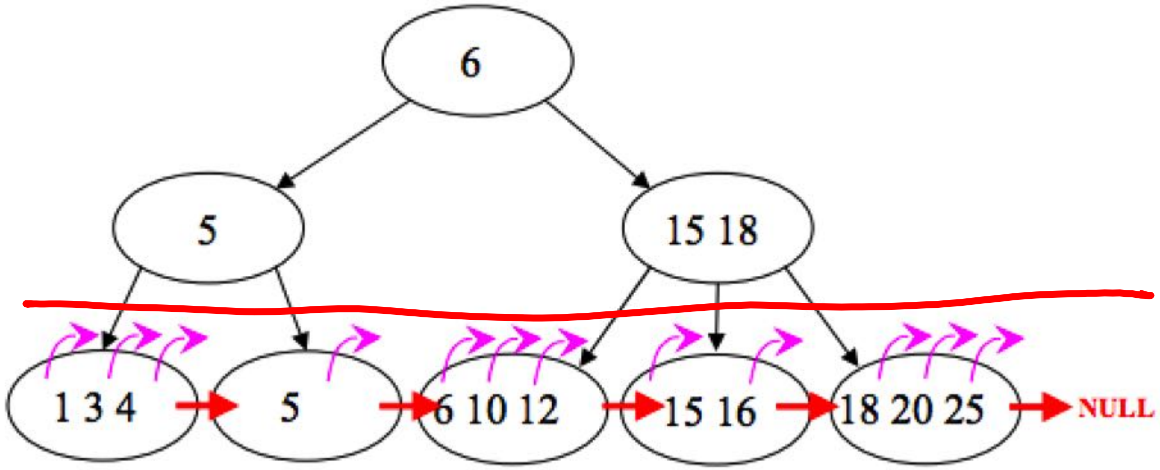
- A B$^+$-tree can be viewed as a B-tree in which each node contains only keys (not pairs), and to which an additional level is added at the bottom with linked leaves

B+-tree maximizes the branching factor of internal nodes!

B-tree of order 4

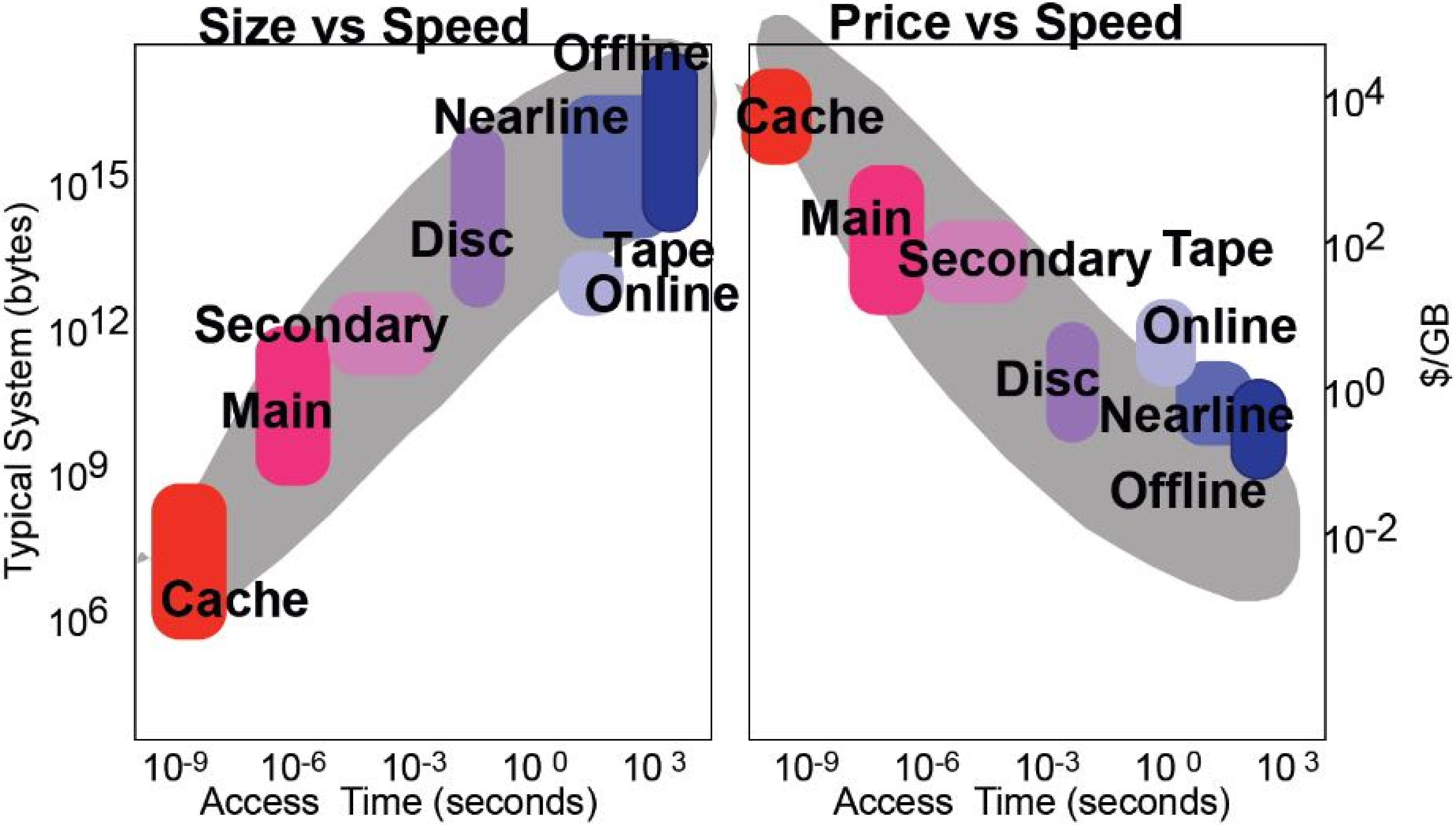


B$^+$-tree of order 4



Source: https://stackoverflow.com/questions/870218/differences-between-b-trees-and-b-trees

Memory hierarchy



Source: "Long Term Storage Trends and You", Jim Gray, 2006: http://jimgray.azurewebsites.net/talks/

Fun with PostgreSQL
Index selection

Recap: Indexes or indices

- Primary mechanism to make queries run faster

- Index on attribute R.A:
  - Creates additional persistent data structure stored with the database
  - Can dramatically speed up certain operations:
    - Find all R tuples where R.A = v
    - Find all R and S tuples where R.A = S.B
    - Find all R tuples where R.A > v (sometimes, depending on index type)

Recap: Index

- A (possibly separate) file, that allows fast access to records in the data file given a search key

again different from "key"!

- The index contains (key, value) pairs:
  - The key = an attribute value
  - The value = either a pointer to the record, or the record itself

Recap: Index classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
  - Unclustered = records close in index may be far in data

- **Primary/secondary**
  - Primary = is over attributes that include the primary key
  - Secondary = otherwise

- Organization: B+ tree or Hash table

Clustered/Unclustered

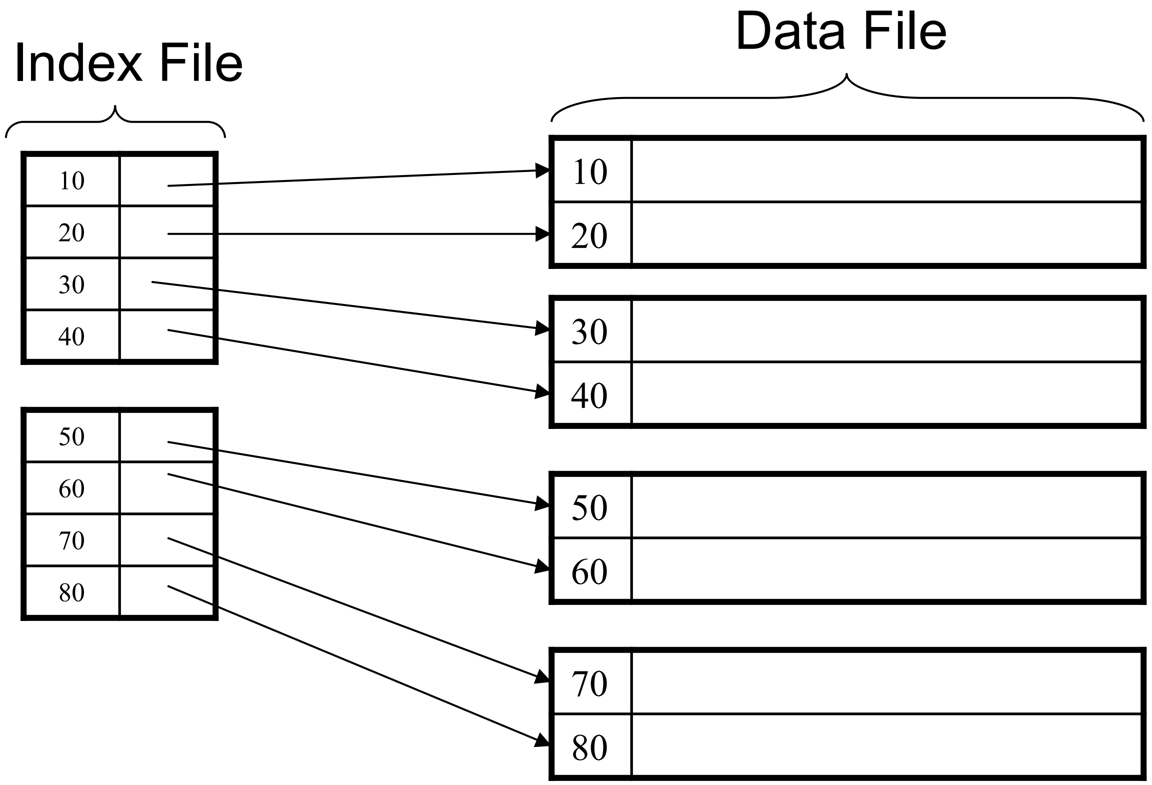
- **Clustered**
  - Index determines the location of indexed records
  - Typically, clustered index is one where values are data records (but not necessary)

- **Unclustered**
  - Index cannot reorder data, does not determine data location
  - In these indexes: value = pointer to data record

```
CLUSTER tableName USING indexName
```

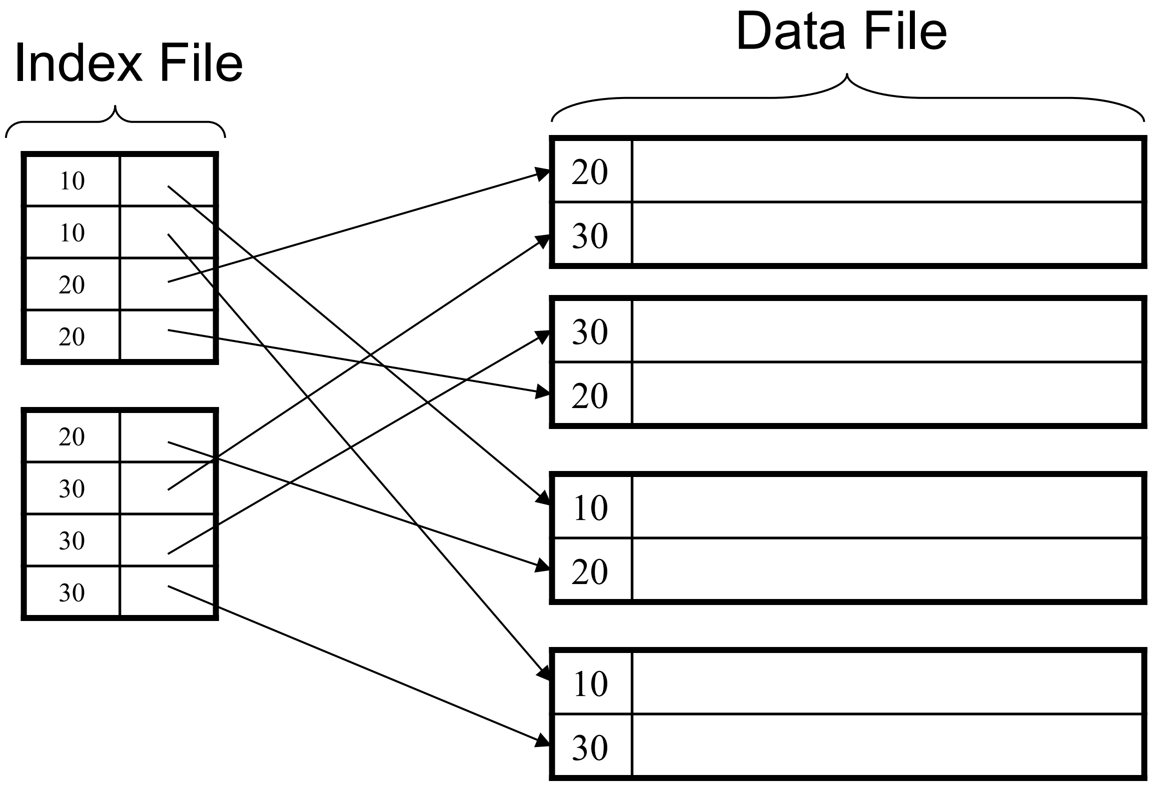
Recap: Clustered index

- File is sorted on the index attribute
- Only one per table

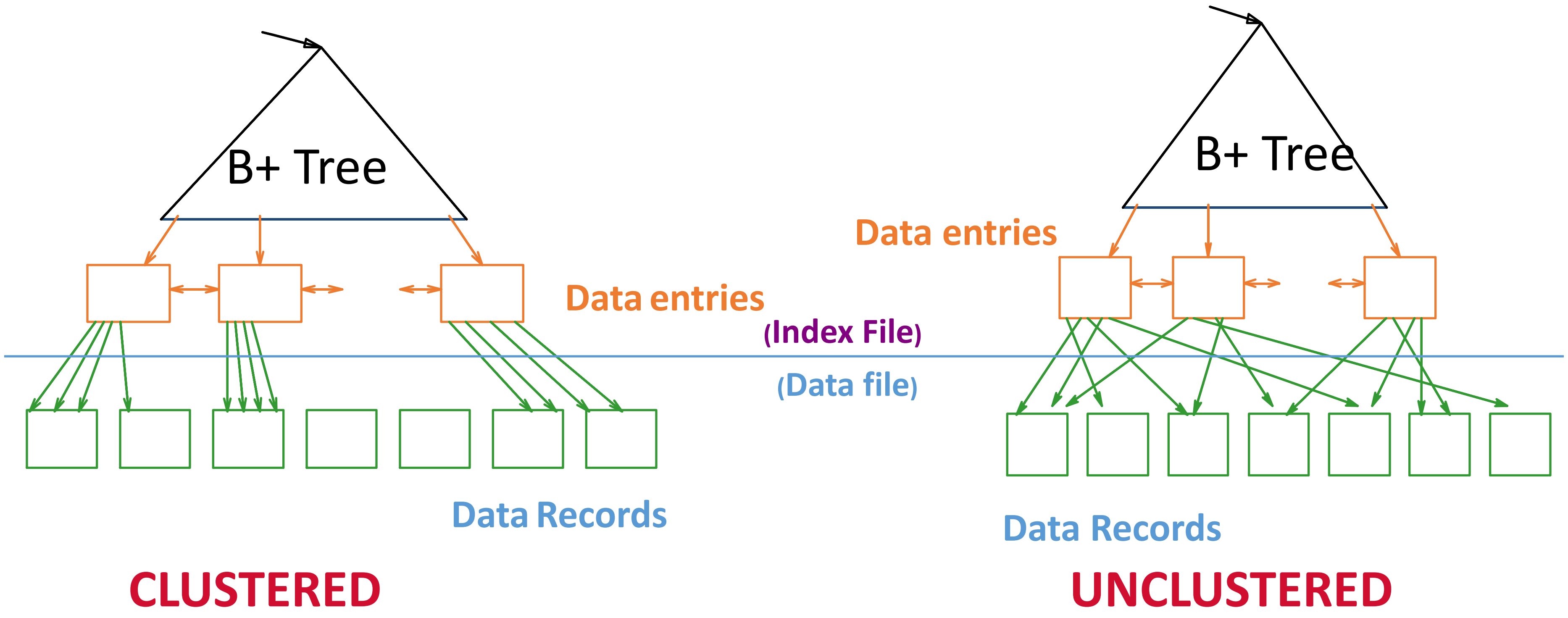


Recap: Unclustered index

- Several per table



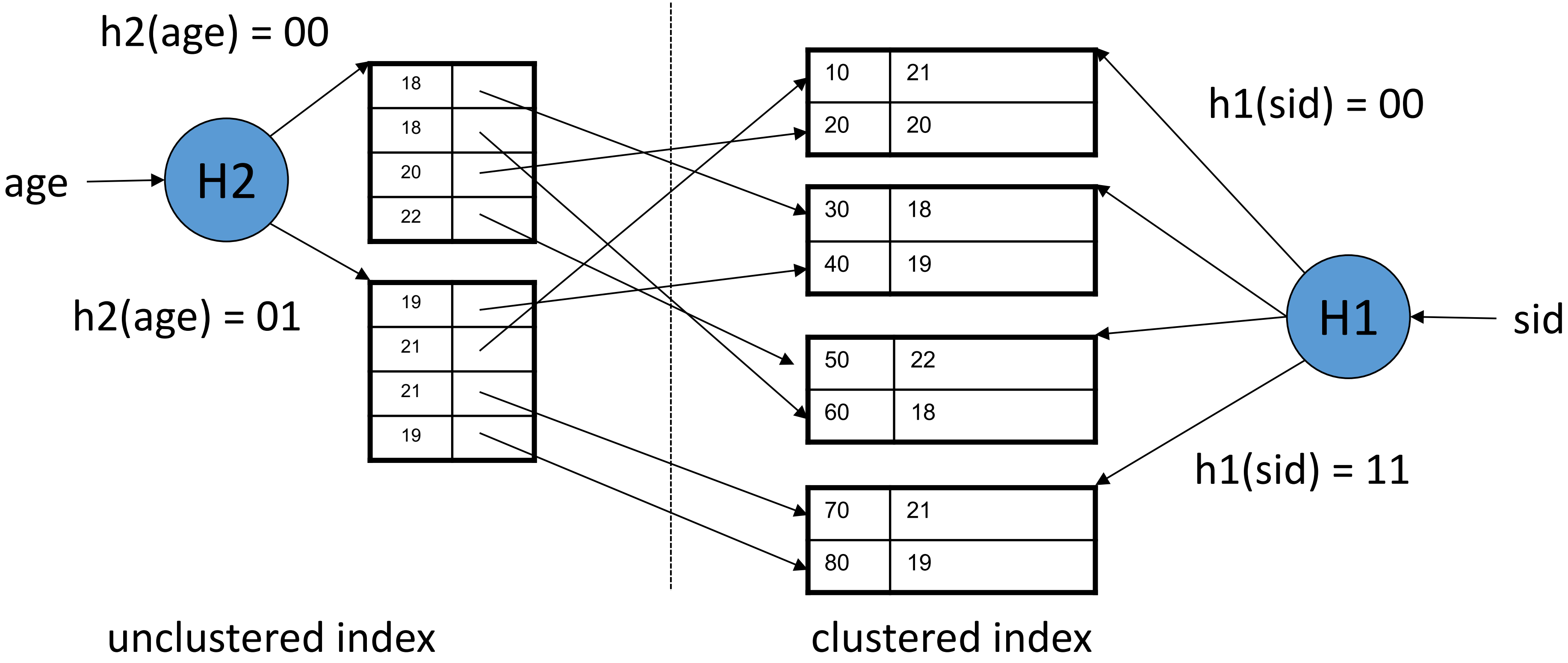
Recap: Clustered vs. unclustered index



More commonly, in a clustered B+ Tree index,
data entries are data records

Hash-based index

Good for point queries but not range queries



Hash Table v.s. B+ tree

- B-tree search:
  - O(logn)
  - Range and equality queries
- Hash search:
  - O(1)
  - Equality only

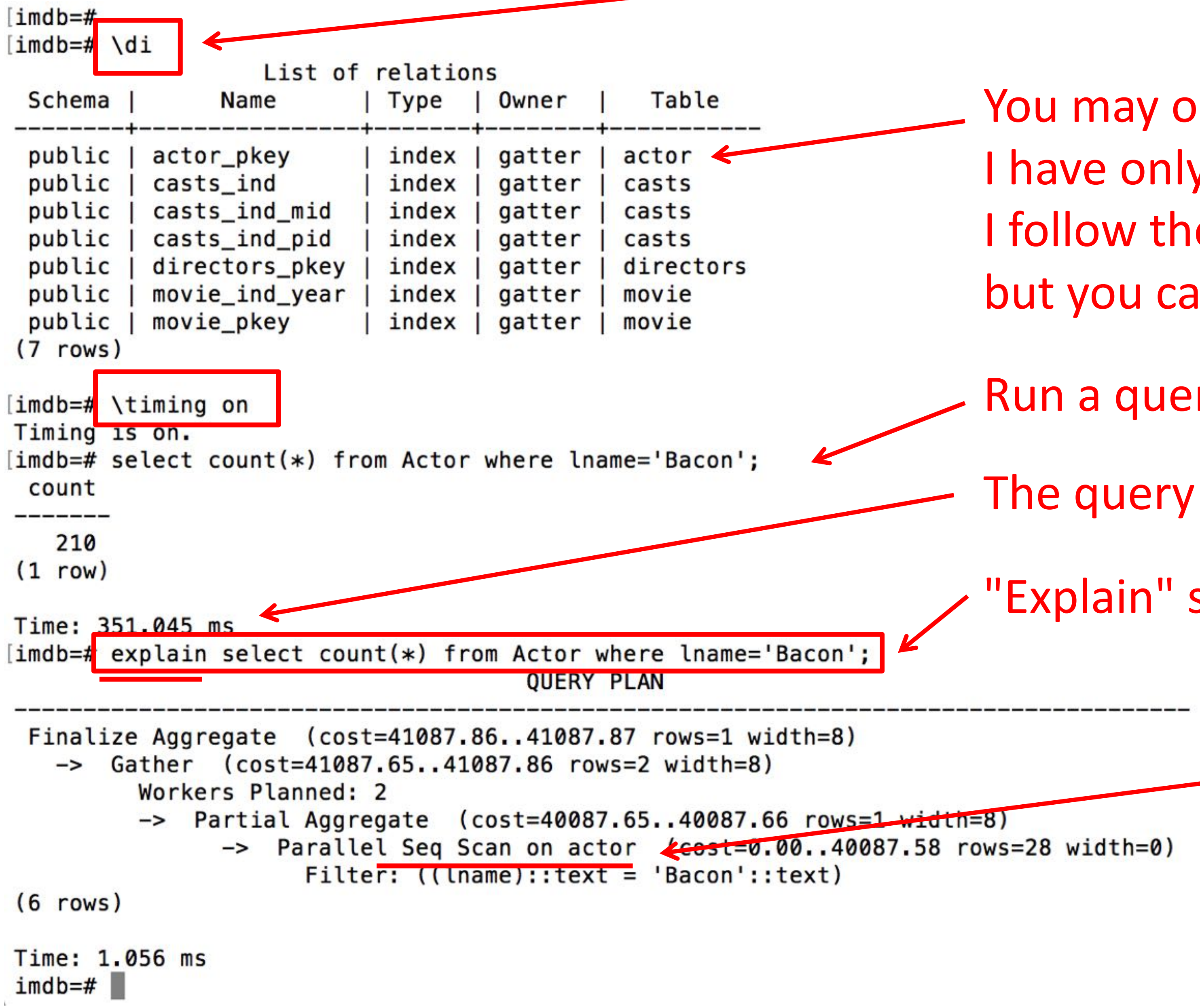
```
CREATE INDEX indexName
ON          tableName
USING       hash(column)
```

- Rule 1: always use a B+ tree ☺
- Rule 2: use a Hash table on K when:
  - There is a very important selection query on equality (WHERE K=?), and no range queries
  - You know that the optimizer uses a nested loop join where K is the join attribute of the inner relation; we will look at this later in more detail ☺

Practice

- Start Postgres and connect to your IMDB database
- Type: `\timing on`
  - Now postgres will report the running time for your queries
- Check for any existing indexes: `\di`
  - Postgres automatically creates indexes on primary keys

Via command line (1/2)



```
[imdb=#
[imdb=# \di
                  List of relations
 Schema |      Name       | Type  | Owner  |   Table
--------+-----------------+-------+--------+-----------
 public | actor_pkey      | index | gatter | actor
 public | casts_ind       | index | gatter | casts
 public | casts_ind_mid   | index | gatter | casts
 public | casts_ind_pid   | index | gatter | casts
 public | directors_pkey  | index | gatter | directors
 public | movie_ind_year  | index | gatter | movie
 public | movie_pkey      | index | gatter | movie
(7 rows)

[imdb=# \timing on
Timing is on.
[imdb=# select count(*) from Actor where lname='Bacon';
 count
-------
   210
(1 row)

Time: 351.045 ms
[imdb=# explain select count(*) from Actor where lname='Bacon';
                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=41087.86..41087.87 rows=1 width=8)
   ->  Gather  (cost=41087.65..41087.86 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=40087.65..40087.66 rows=1 width=8)
               ->  Parallel Seq Scan on actor  (cost=0.00..40087.58 rows=28 width=0)
                     Filter: ((lname)::text = 'Bacon'::text)
(6 rows)

Time: 1.056 ms
imdb=# 
```

See existing indexes

You may or may not have existing indexes.
I have only one on the actor table.
I follow the naming scheme <table_attribute>,
but you can choose the names

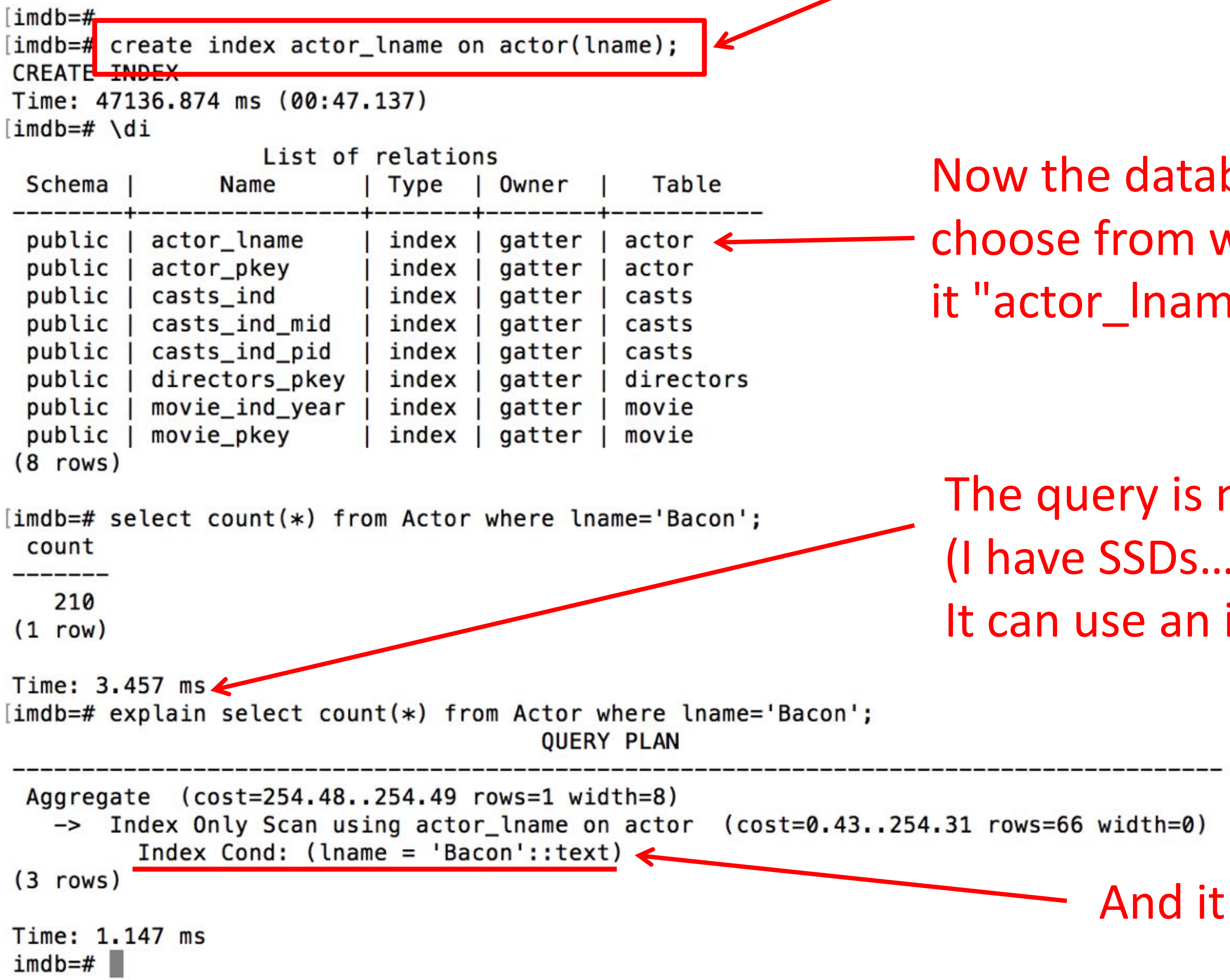
Run a query that filters on lname

The query takes 351 ms

"Explain" shows the query plan

The query plan scans the whole
actor table; that takes time…

Via command line (2/2)



```
[imdb=#
[imdb=# create index actor_lname on actor(lname);
CREATE INDEX
Time: 47136.874 ms (00:47.137)
[imdb=# \di
                    List of relations
 Schema |      Name       | Type  | Owner  |   Table
--------+-----------------+-------+--------+-----------
 public | actor_lname     | index | gatter | actor
 public | actor_pkey      | index | gatter | actor
 public | casts_ind       | index | gatter | casts
 public | casts_ind_mid   | index | gatter | casts
 public | casts_ind_pid   | index | gatter | casts
 public | directors_pkey  | index | gatter | directors
 public | movie_ind_year  | index | gatter | movie
 public | movie_pkey      | index | gatter | movie
(8 rows)

[imdb=# select count(*) from Actor where lname='Bacon';
 count
-------
   210
(1 row)

Time: 3.457 ms
[imdb=# explain select count(*) from Actor where lname='Bacon';
                                         QUERY PLAN
---------------------------------------------------------------------------------------
 Aggregate  (cost=254.48..254.49 rows=1 width=8)
   ->  Index Only Scan using actor_lname on actor  (cost=0.43..254.31 rows=66 width=0)
         Index Cond: (lname = 'Bacon'::text)
(3 rows)

Time: 1.147 ms
imdb=# █
```

Let's create an index on lname. That takes 47 sec

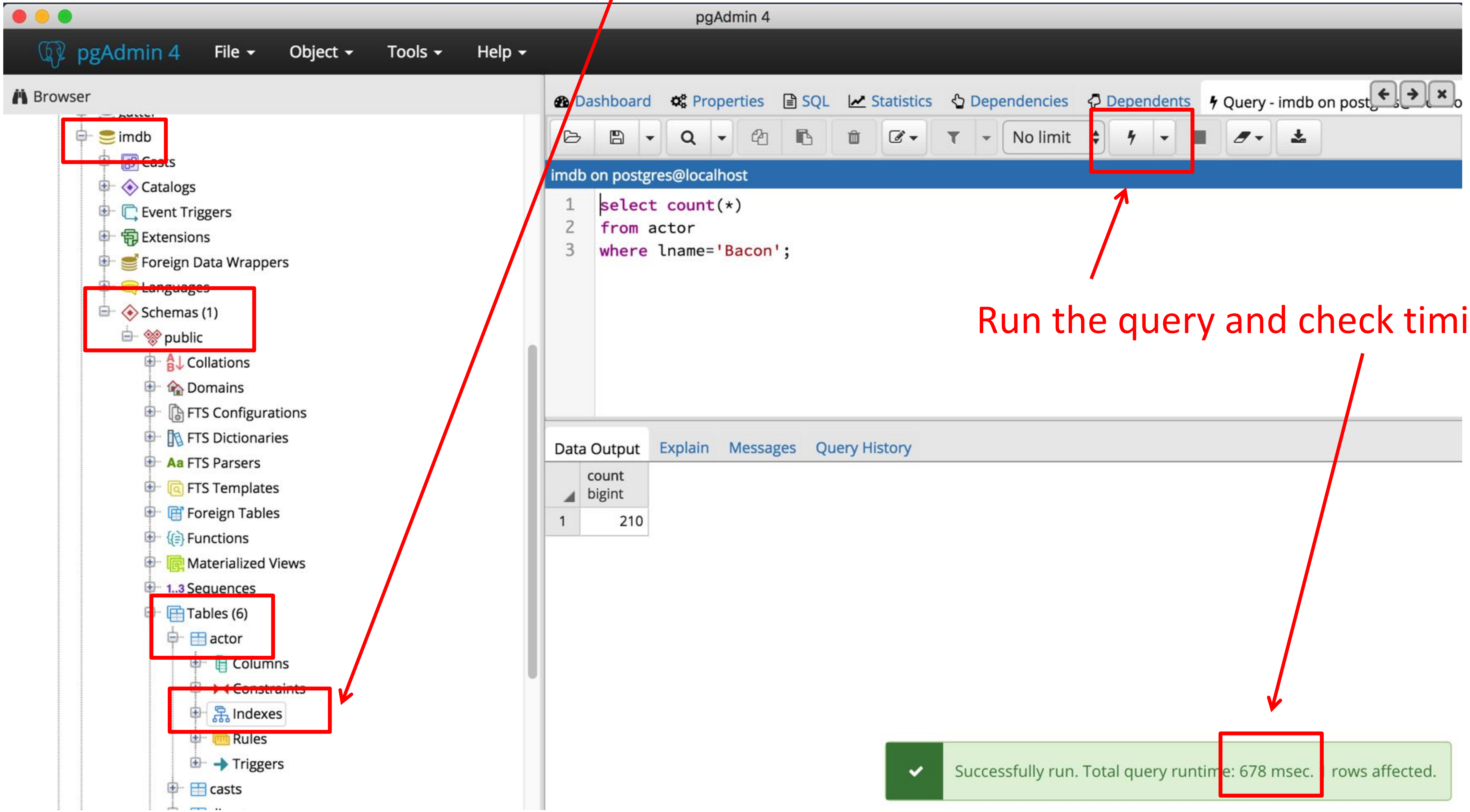
Now the database has an additional index it can choose from when answering your query. I called it "actor_lname"

The query is now 100 times faster: 3.5 ms
(I have SSDs…).
It can use an index to lookup 'Bacon'

And it does ☺

Via PgAdmin (1/8)

Navigate towards actor indexes run the query



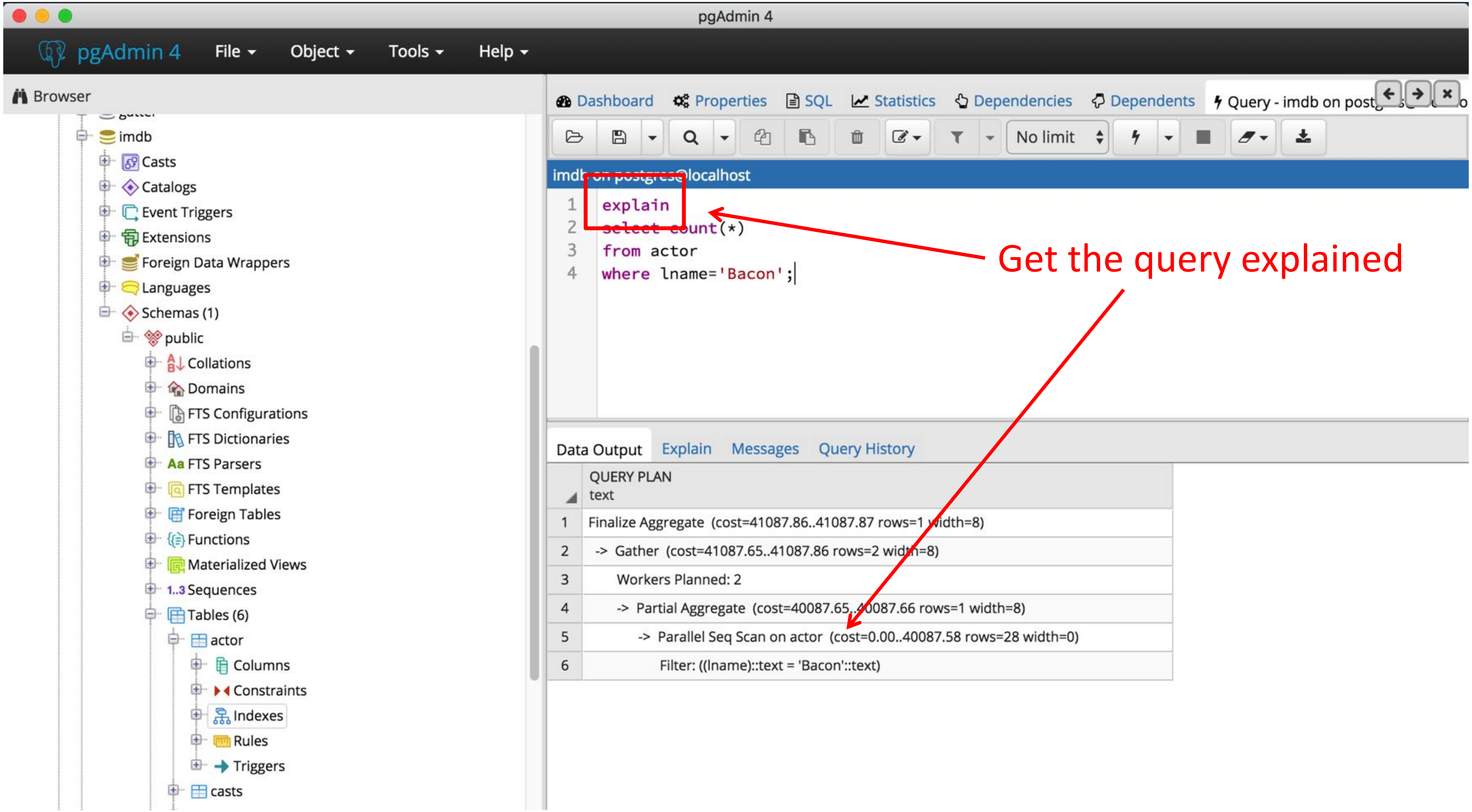
```
select count(*)
from actor
where lname='Bacon';
```

```
count
bigint
1    210
```

Successfully run. Total query runtime: 678 msec. rows affected.

Run the query and check timing

Via PgAdmin (2/8)

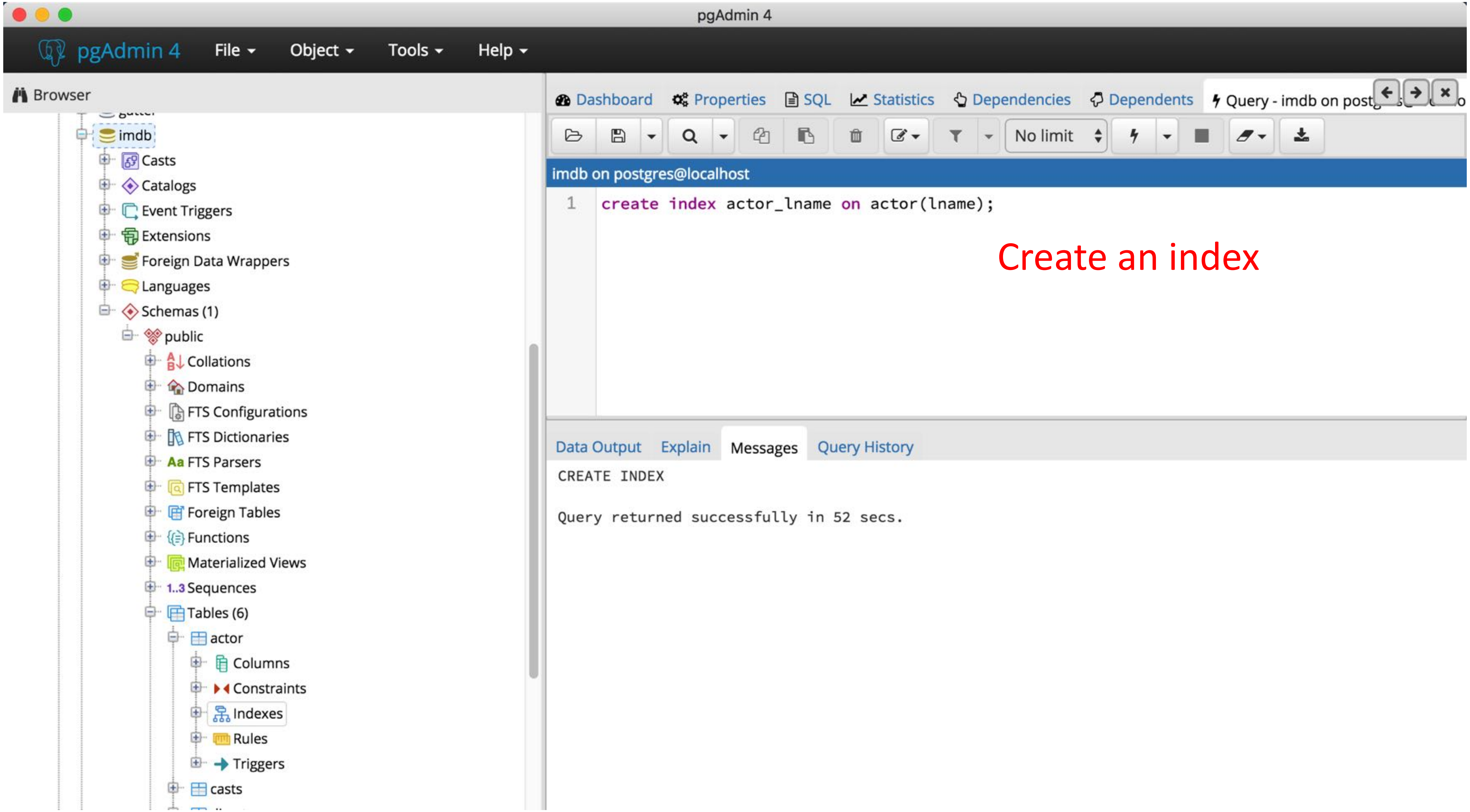


```
explain
select count(*)
from actor
where lname='Bacon';
```

Get the query explained

```
QUERY PLAN
text
1  Finalize Aggregate  (cost=41087.86..41087.87 rows=1 width=8)
2    -> Gather  (cost=41087.65..41087.86 rows=2 width=8)
3         Workers Planned: 2
4         -> Partial Aggregate  (cost=40087.65..40087.66 rows=1 width=8)
5              -> Parallel Seq Scan on actor  (cost=0.00..40087.58 rows=28 width=0)
6                   Filter: ((lname)::text = 'Bacon'::text)
```

Via PgAdmin (3/8)



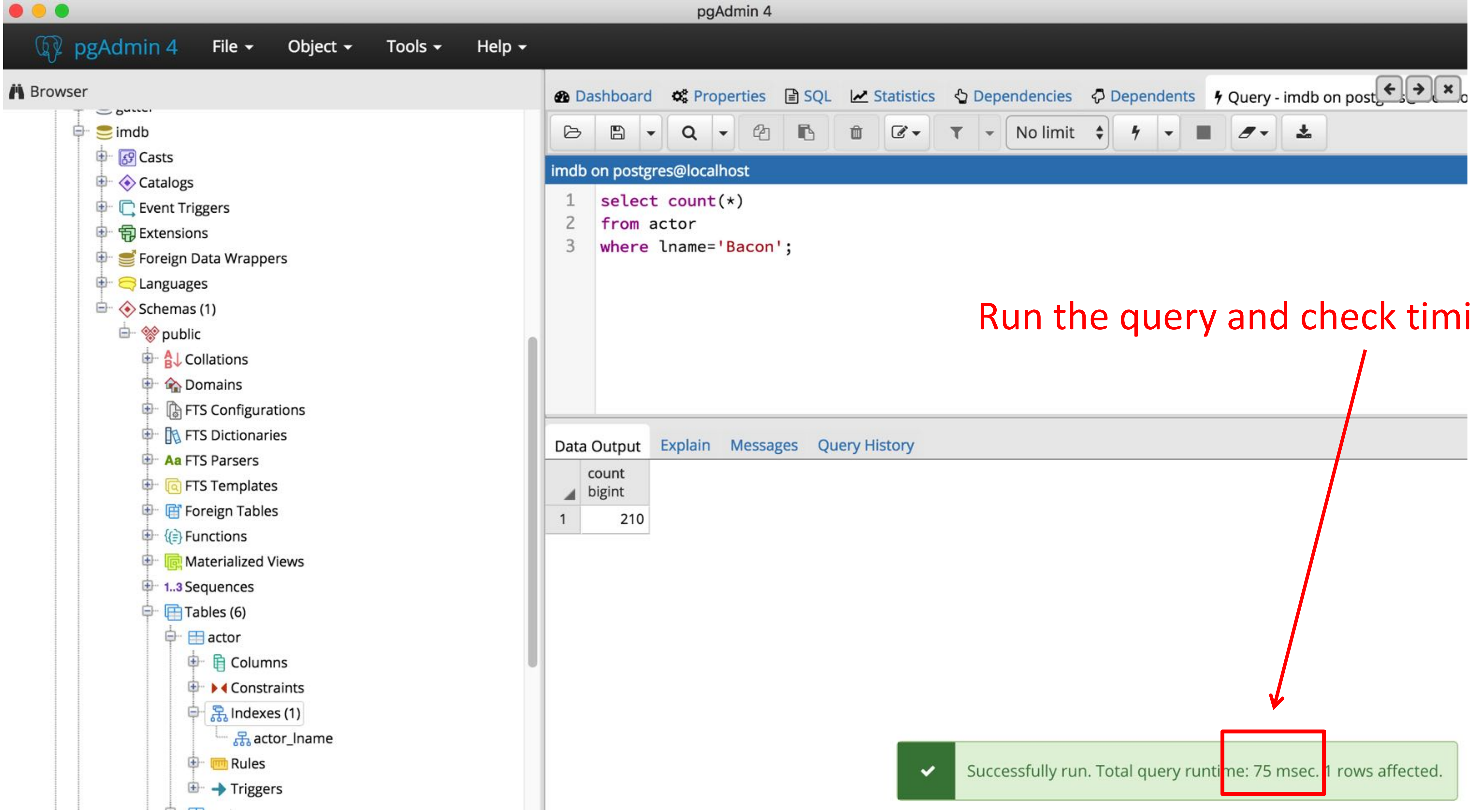
```
create index actor_lname on actor(lname);
```

Create an index

```
CREATE INDEX

Query returned successfully in 52 secs.
```

Via PgAdmin (4/8)



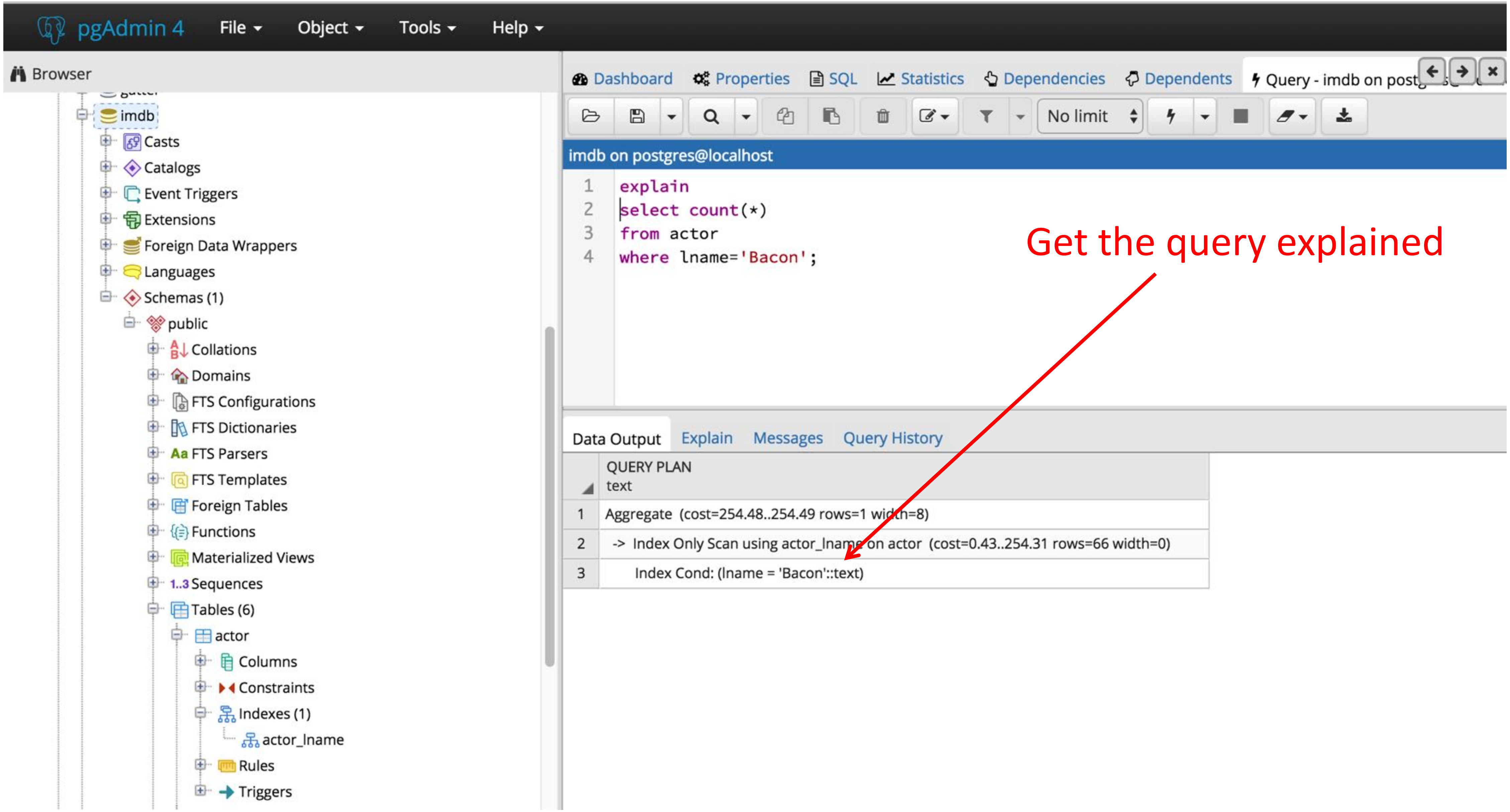
```
select count(*)
from actor
where lname='Bacon';
```

	count bigint
1	210

Successfully run. Total query runtime: 75 msec. 1 rows affected.

Run the query and check timing

Via PgAdmin (5/8)

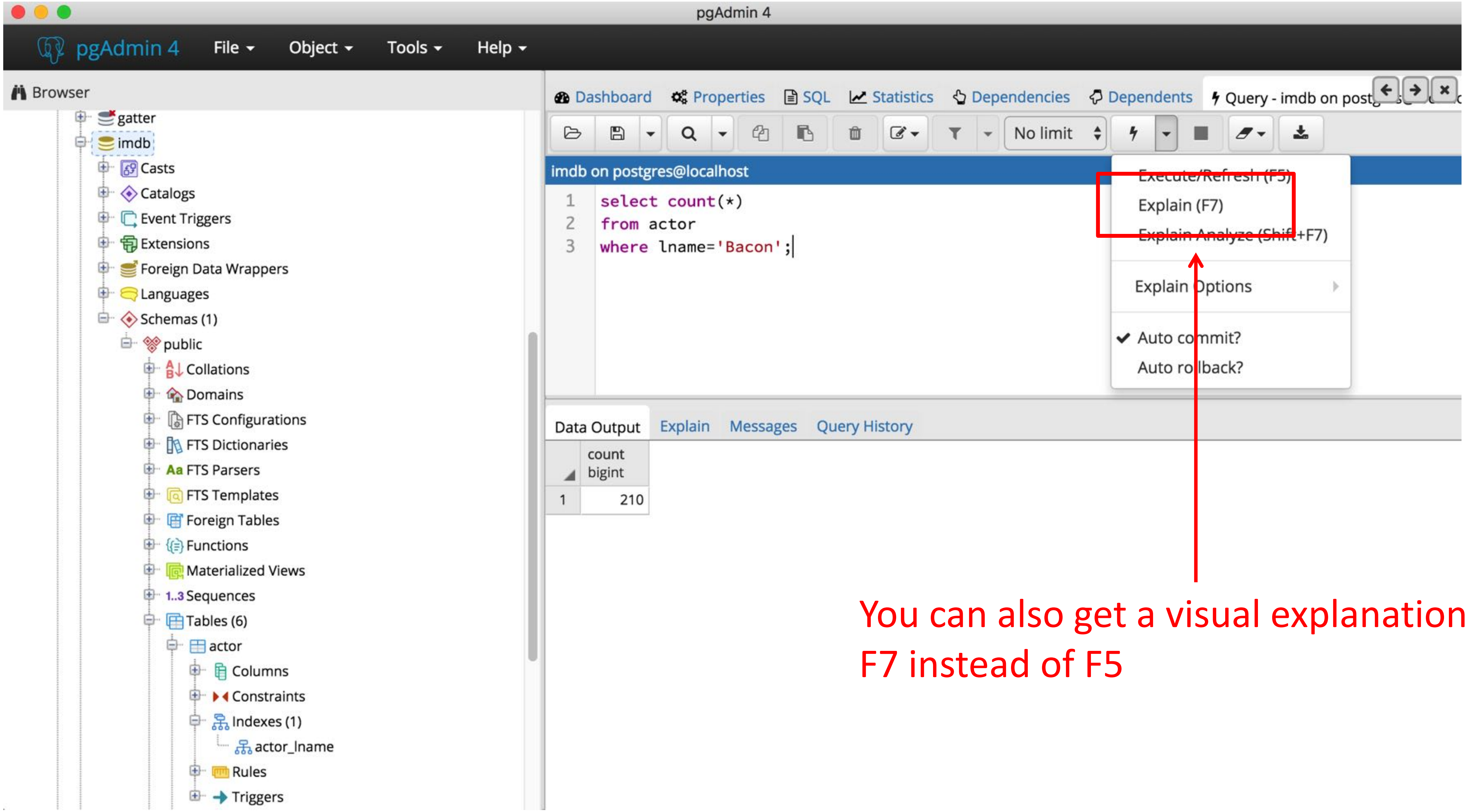


```
explain
select count(*)
from actor
where lname='Bacon';
```

QUERY PLAN text
Aggregate (cost=254.48..254.49 rows=1 width=8)
-> Index Only Scan using actor_lname on actor (cost=0.43..254.31 rows=66 width=0)
Index Cond: (lname = 'Bacon'::text)

Get the query explained

Via PgAdmin (6/8)

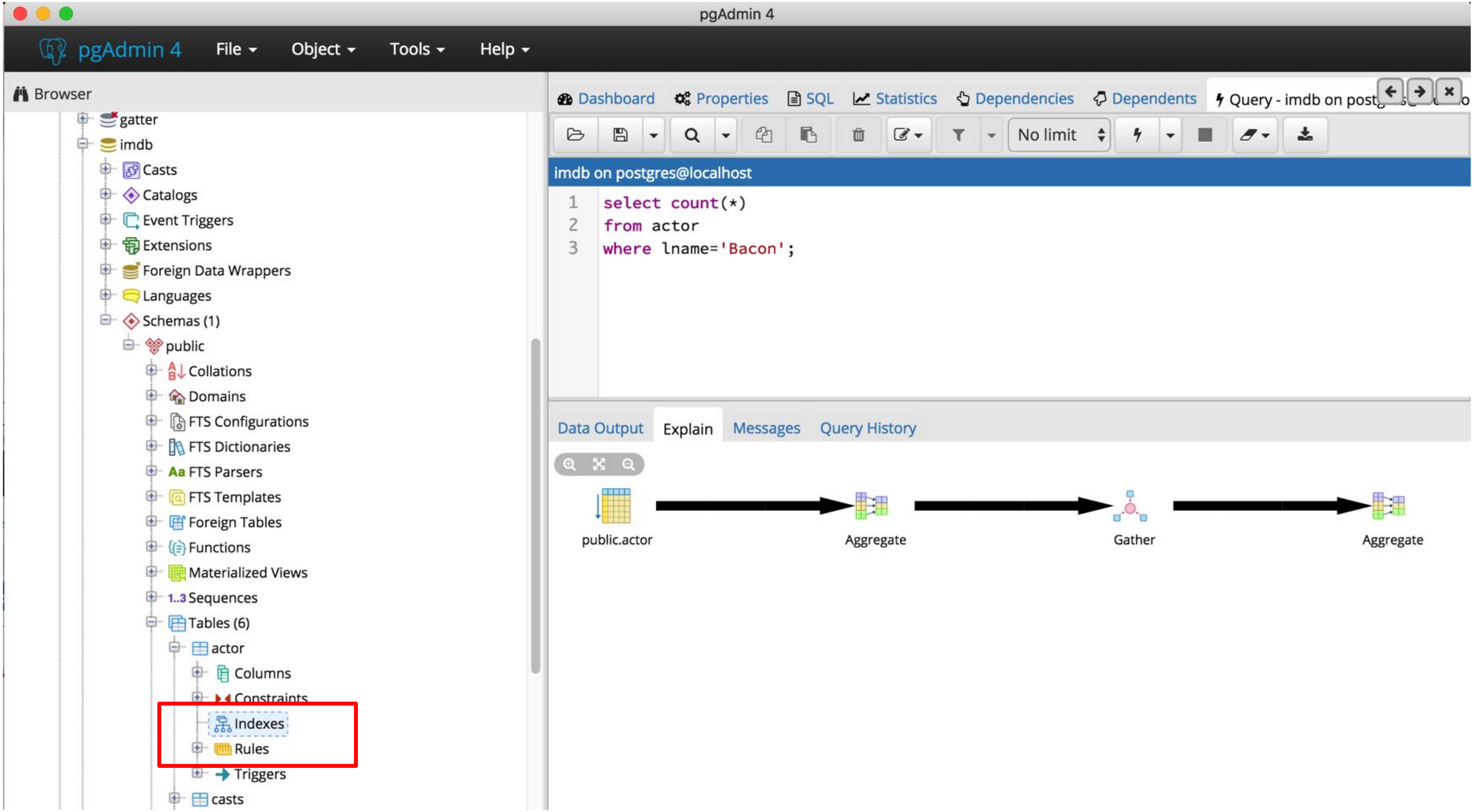


```
select count(*)
from actor
where lname='Bacon';
```

	count bigint
1	210

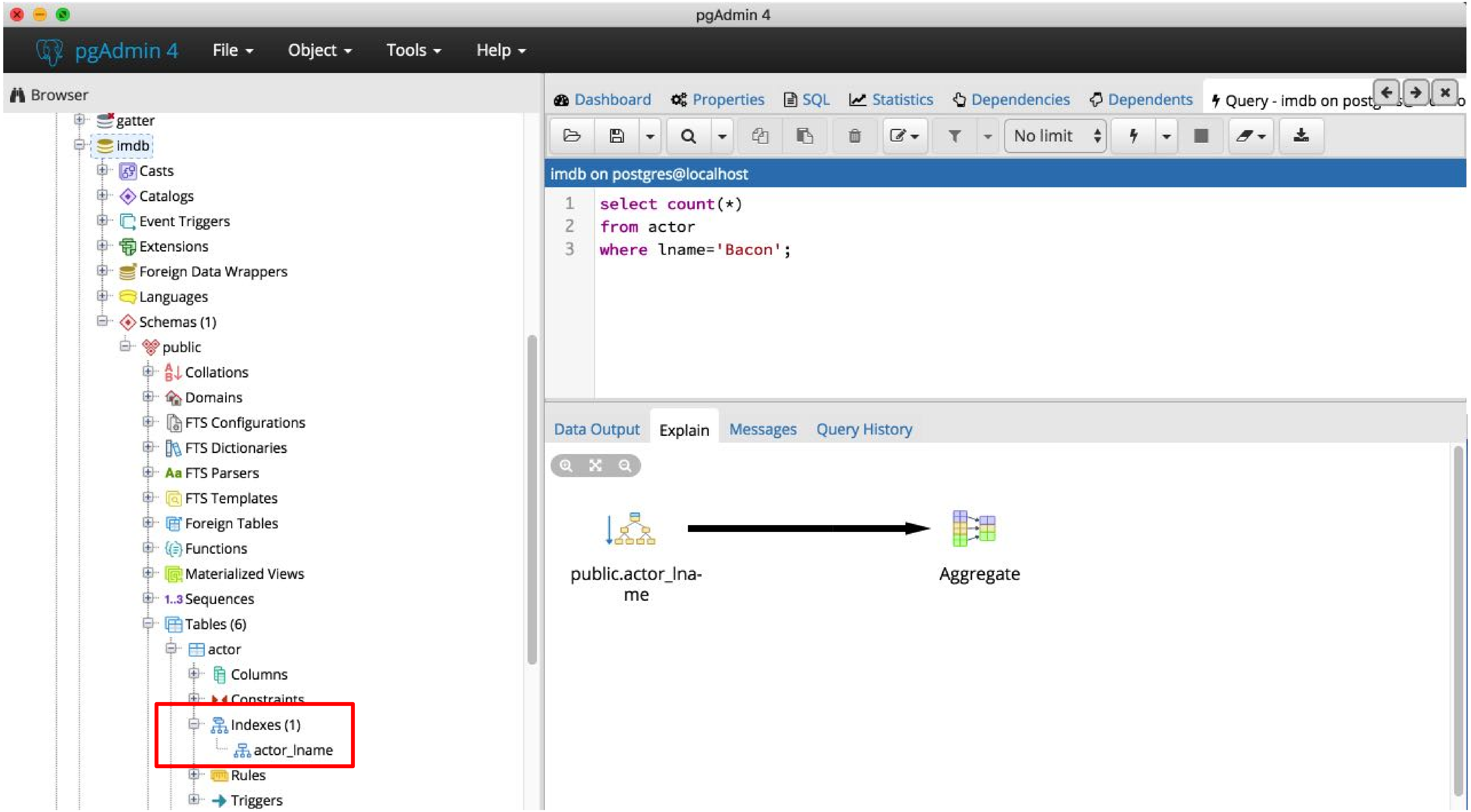
You can also get a visual explanation
F7 instead of F5

Via PgAdmin (7/8)



```sql
select count(*)
from actor
where lname='Bacon';
```

Via PgAdmin (8/8)



```
select count(*)
from actor
where lname='Bacon';
```

Practice

```
SELECT *
FROM   Actor
WHERE  lname = 'Bacon'
```

How long does it take to run?

Let’s see how the query is executed:

```
EXPLAIN
SELECT *
FROM   Actor
WHERE  lname = 'Bacon'
```

Introduce indexes

```
CREATE INDEX actorLName
ON       Actor(lname)
```

```
SELECT  *
FROM    Actor
WHERE   lname = 'Bacon'
```

How long does it take now?

Let’s see how the query is executed this time:

```
EXPLAIN
SELECT  *
FROM    Actor
WHERE   lname = 'Bacon'
```

Practice

Look at the indexes on table Actor: \d Actor

Let’s get execution plans for different queries:

```
EXPLAIN
SELECT  *
FROM    Actor
WHERE   lname = ‘Bacon’ AND id > 50000
```

```
EXPLAIN
SELECT  *
FROM    Actor
WHERE   lname = ‘Bacon’ AND id = 50000
```

Indexes and joins

```
SELECT  C.role
FROM    Actor A, Casts C
WHERE   lname = ‘Bacon’ AND A.id = C.pid
```

How long does it take to run?

Let’s see how the query is executed:

```
EXPLAIN
SELECT  C.role
FROM    Actor A, Casts C
WHERE   lname = ‘Bacon’ AND A.id = C.pid
```

EXPLAIN

How the join happens

How ‘Casts’ is accessed

```
 QUERY PLAN
-----------------------------------------------------------------------------------------
 Hash Join  (cost=118.44..238227.16 rows=233 width=12)
   Hash Cond: (c.pid = a.id)
   ->  Seq Scan on casts c  (cost=0.00..195184.47 rows=11445847 width=16)
   ->  Hash  (cost=117.96..117.96 rows=38 width=4)
         ->  Index Scan using actorlname on actor a  (cost=0.00..117.96 rows=38 width=4)
               Index Cond: ((lname)::text = 'Bacon'::text)
```

How ‘Actor’ is accessed

Indexes and joins

```
CREATE INDEX castActorId
ON      Casts(pid)
```

```
SELECT  C.role
FROM    Actor A, Casts C
WHERE   lname = ‘Bacon’ AND A.id = C.pid
```

How long does it take now?

Let’s see how the query is executed this time:

```
EXPLAIN
SELECT  C.role
FROM    Actor A, Casts C
WHERE   lname = ‘Bacon’ AND A.id = C.pid
```

EXPLAIN

Different type of join

```
  QUERY PLAN
-----------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..1272.60 rows=233 width=12)
   ->  Index Scan using actorlname on actor a  (cost=0.00..117.96 rows=38 width=4)
         Index Cond: ((lname)::text = 'Bacon'::text)
   ->  Index Scan using castactor on casts c  (cost=0.00..28.53 rows=186 width=16)
         Index Cond: (pid = a.id)
```

Both indexes are used

Joins

1) Nested Loop Join
2) Sort-Merge Join
3) Hash Join

1. Nested Loop Joins

a) Recap Joins
b) Sort-Merge Join
c) Hash Join

What we will learn next

- RECAP: Joins

- Nested Loop Join (NLJ)

- Block Nested Loop Join (BNLJ)

- Index Nested Loop Join (INLJ)

Recap: Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

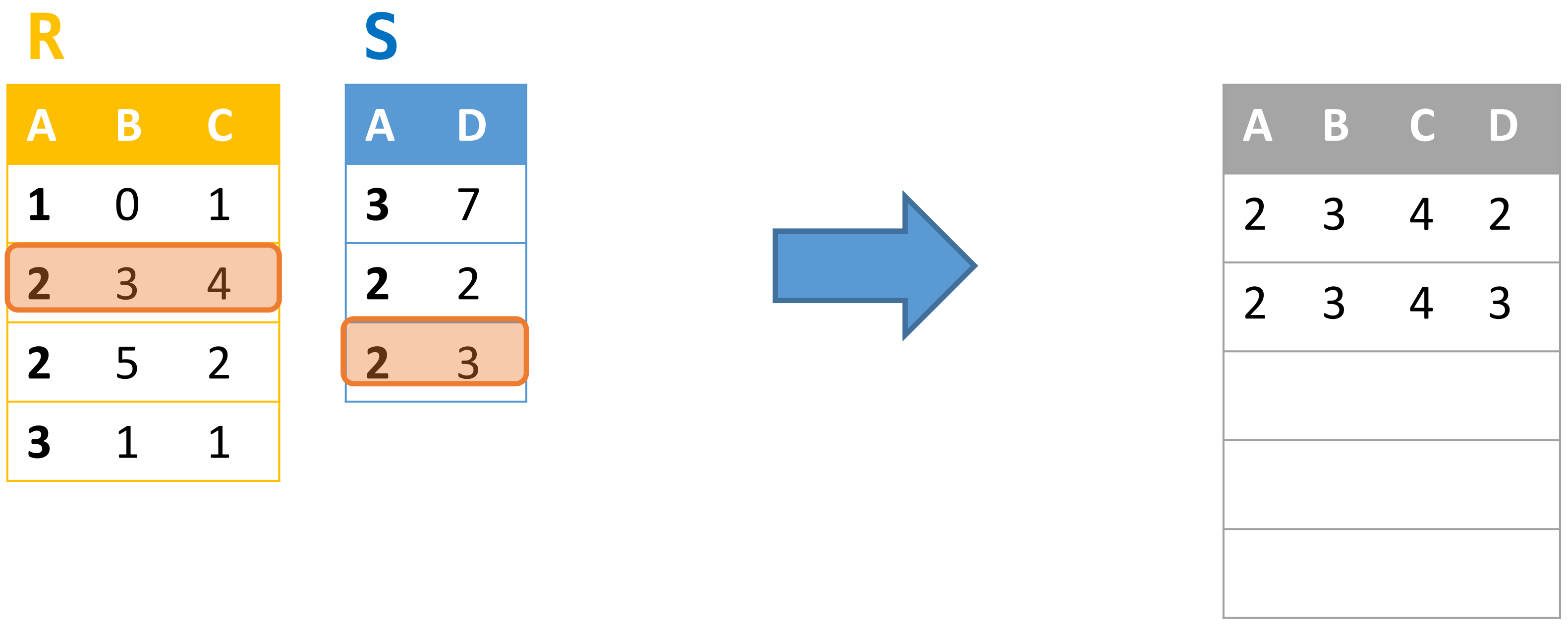
A	B	C	D
2	3	4	2

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT  R.A,B,C,D
FROM    R, S
WHERE   R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$



R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

<u>Example:</u> Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2

Joins: Example

$\mathbf{R} \bowtie S$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT  R.A,B,C,D
FROM    R, S
WHERE   R.A = S.A
```

<u>Example:</u> Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Semantically: A Subset of the Cross Product

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

×

S

A	D
3	7
2	2
2	3

Cross Product

…

Filter by conditions (r.A = s.A)

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Can we actually implement a join in this way?

Notes

- We write **$\mathbf{R} \bowtie \boldsymbol{S}$** to mean join R and S by returning all tuple pairs where all shared attributes are equal ("natural join")

- We write **$\mathbf{R} \bowtie \boldsymbol{S}$ on A** to mean join R and S by returning all tuple pairs where attribute(s) A are equal

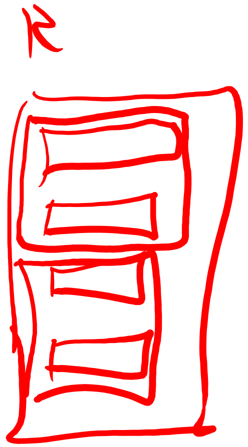
- For simplicity, we'll consider joins on two tables (binary joins) and with equality constraints ("equijoins")

However joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!

Nested Loop Joins

Notes

- We are again considering “IO aware” algorithms: <u>care about disk IO</u>
- Given a relation R, let:
  - T(R) = # of tuples in R
  - P(R) = # of pages in R



$T(R) = 4$

$P(R) = 2$

Recall that we read / write entire pages with disk IO

- Note also that we omit ceilings in calculations… good exercise to put back in!

ceiling(x) = $\lceil x \rceil$ = smallest integer ≥ x

Nested Loop Join (NLJ)

```
Compute $R \bowtie S \; on \; A$:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Nested Loop Join (NLJ)

```
Compute $R \bowtie S\ on\ A$:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R)

1. **Loop over the tuples in R**

Note that our IO cost is based on the number of ***pages*** loaded, not the number of tuples!

Nested Loop Join (NLJ)

```
Compute $R \bowtie S \ on \ A$:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S)

1. Loop over the tuples in R
2. **For every tuple in R, loop over all the tuples in S**

Have to read ***all of S*** from disk for ***every tuple in R!***

Nested Loop Join (NLJ)

```
Compute $R \bowtie S \; on \; A$:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S)

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. **Check against join conditions**

Note that NLJ can handle things other than equality constraints… just check in the *if* statement!

Nested Loop Join (NLJ)

```
Compute $R \bowtie S\ on\ A$:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

<u>Cost:</u>

P(R) + T(R)*P(S) + OUT

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

What would ***OUT*** be if our join condition is trivial (*if TRUE*)?

OUT could be bigger than P(R)*P(S)… but usually not that bad

Nested Loop Join (NLJ)

```
Compute $R \bowtie S$ on $A$:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S) + OUT

What if R ("outer") and S ("inner") switched?

↓

P(***S***) + T(***S***)*P(***R***) + OUT

Outer vs. inner selection makes a huge difference- DBMS needs to know which relation is smaller!

IO-Aware Approach

Block Nested Loop Join (BNLJ)

Notice that our text book and Gradiance use M just for the input buffer and assume 1 extra page for the output

Given ***M*** pages of memory

```
Compute $R \bowtie S \ on \ A$:
  for each M-2 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Cost:

$P(R)$

1. **Load in M-2 pages of R at a time (leaving 1 page each free for S & output)**

Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!

Block Nested Loop Join (BNLJ)

```
Compute $R \bowtie S$ on $A$:
  for each M-2 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Given ***M*** pages of memory

<u>Cost:</u>

$$P(R) + \frac{P(R)}{M-2} P(S)$$

1. Load in M-2 pages of R at a time (leaving 1 page each free for S & output)

2. **For each (M-2)-page segment of R, load each page of S**

Note: Faster to iterate over the *smaller* relation first!

Block Nested Loop Join (BNLJ)

Given ***M*** pages of memory

Cost:

$$P(R) + \frac{P(R)}{M-2} P(S)$$

```
Compute $R \bowtie S \text{ on } A$:
  for each M–2 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

1. Load in M-2 pages of R at a time (leaving 1 page each free for S & output)
2. For each (M-2)-page segment of R, load each page of S
3. **Check against the join conditions**

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

```
Compute R ⋈ S on A:
  for each M−2 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Again, *OUT* could be bigger than P(R)*P(S)… but usually not that bad

Given ***M*** pages of memory

Cost:

$$P(R) + \frac{P(R)}{M-2} P(S) + \text{OUT}$$

1. Load in M-2 pages of R at a time (leaving 1 page each free for S & output)
2. For each (M-2)-page segment of R, load each page of S
3. Check against the join conditions
4. **Write out**

BNLJ vs. NLJ: Benefits of IO Aware

- In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S
  - We only read all of S from disk for ***every (M-2)-page segment of R***!
  - Still the full cross-product, but more done only *in memory*

NLJ

$$P(R) + T(R)*P(S) + \text{OUT}$$

→

BNLJ

$$P(R) + \frac{P(R)}{M-2}P(S) + \text{OUT}$$

BNLJ is faster by roughly $\frac{(M-2)T(R)}{P(R)}$!

BNLJ vs. NLJ: Benefits of IO Aware

- Example:
  - R: 500 pages
  - S: 1000 pages
  - 100 tuples / page
  - We have 102 pages of memory (M = 102), one of which is for output buffer

Ignoring cost of OUT here…

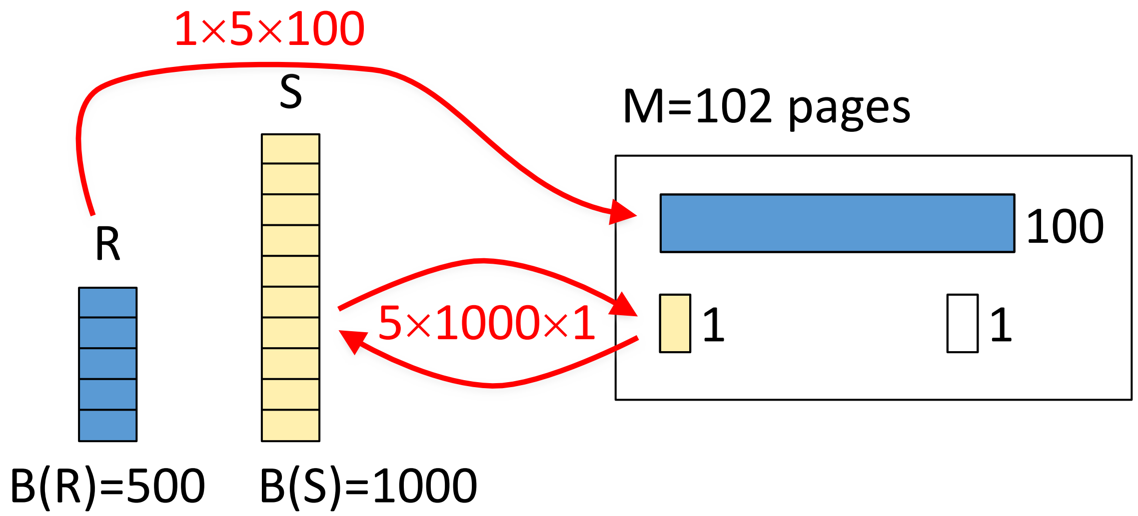
- NLJ: Cost = 500 + **500*100***1000 = **50 Million IOs ~= <u>140 hours</u>**

- BNLJ: Cost = 500 + $\mathbf{\frac{500}{100}}$*1000 = **5,500 IOs ~< <u>1 min</u>**

assuming 10 ms per IO

A very real difference from a small change in the algorithm!

NLJ: Order of tables matters


1×5×100

S

M=102 pages

R

100

5×1000×1

1

1

B(R)=500

B(S)=1000

Ignoring output cost

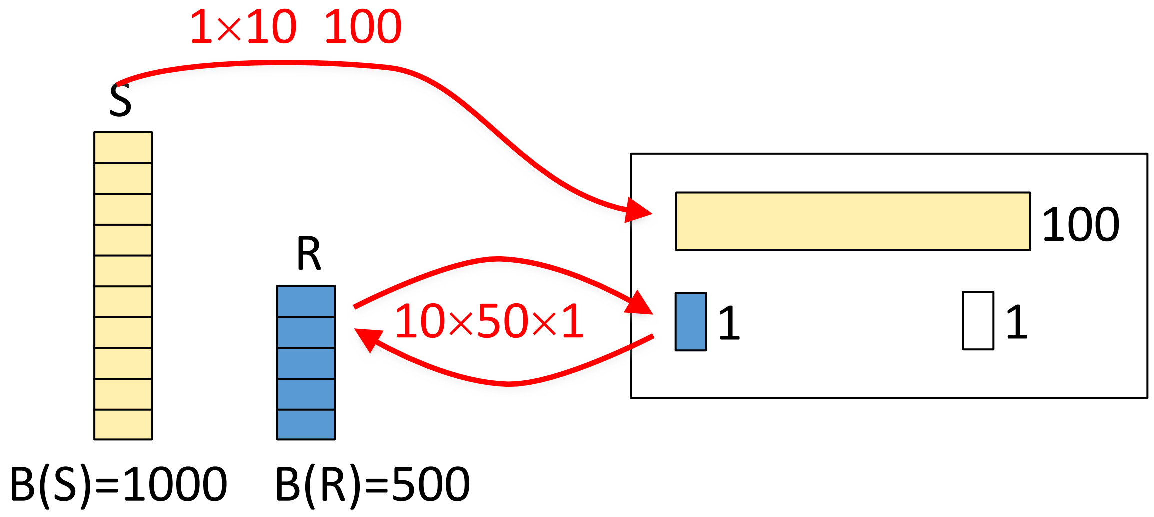
Cost R: 500

Cost S: 5,000 = 5×1,000

SUM: **5,500**

B(R) + B(R)/(M-2)×B(S)

500 + (500/100)×1,000 = **5,500**


1×10 100

S

R

100

10×50×1

1

1

B(S)=1000

B(R)=500

Cost S: 1,000

Cost R: 5,000 = 10×500

SUM: **6,000**

B(S) + B(R)/(M-2)×B(S)

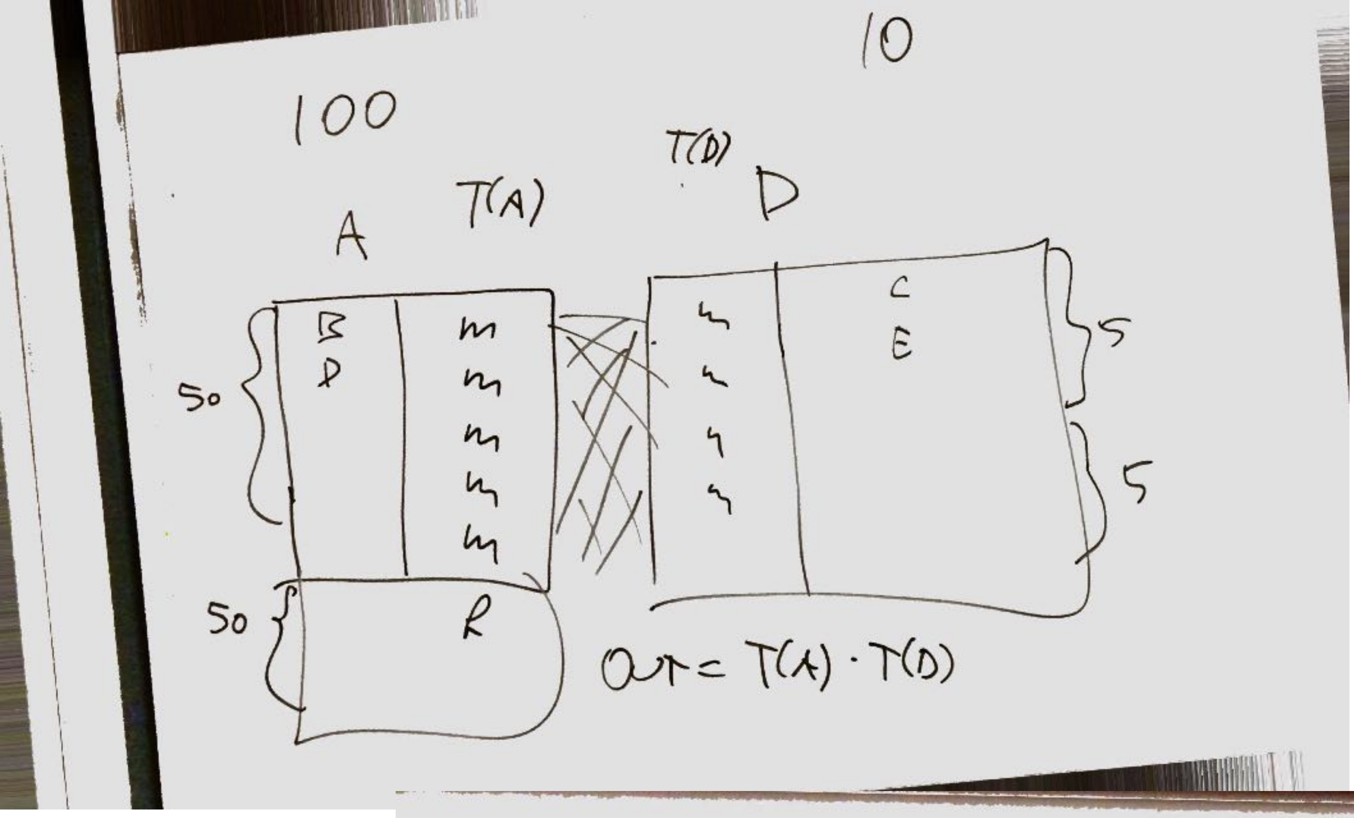
1000 + (1,000/100)×500 = **6,000**

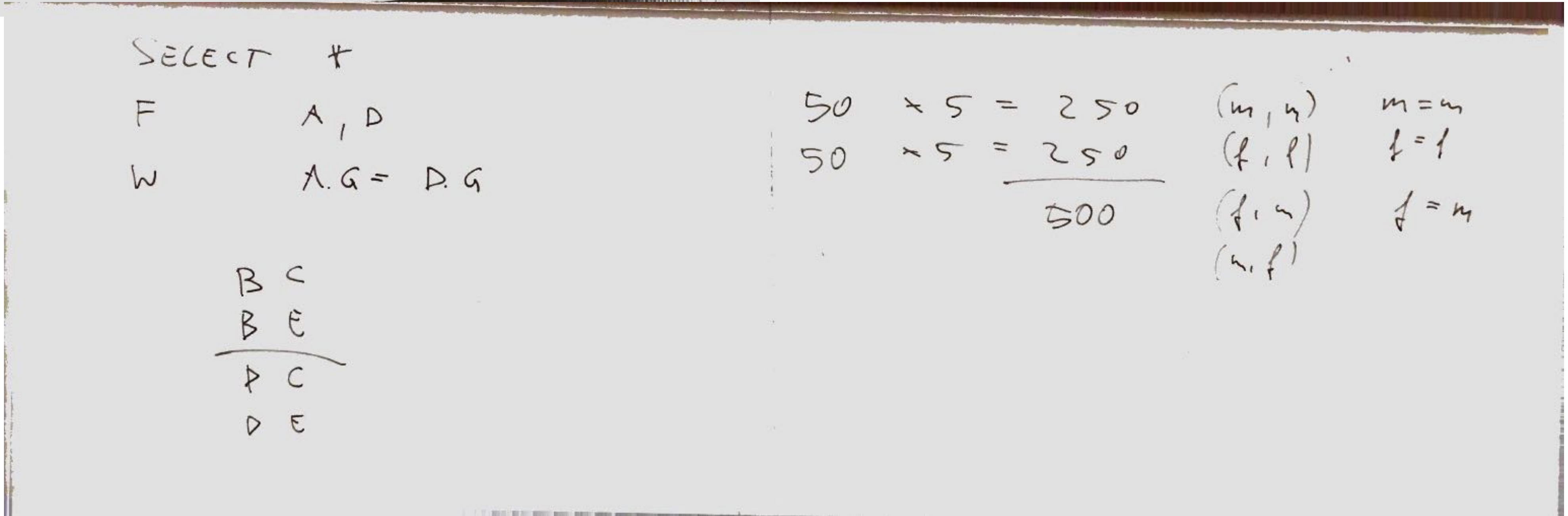
Variant of Example 15.4 from "Cow book" (Ramakrishan, Gehrke, Database management systems, 2003)

Whiteboard example

- Assume the table actor has 100 entries and the table director has 10 entries. Half of the actors and directors are female.
- How large is the result of following query?

```
SELECT *
FROM   Actor A, Director D
WHERE  A.gender = D.gender
```





SELECT *

F A, D

W A.G = D.G

B C
B E
D C
D E

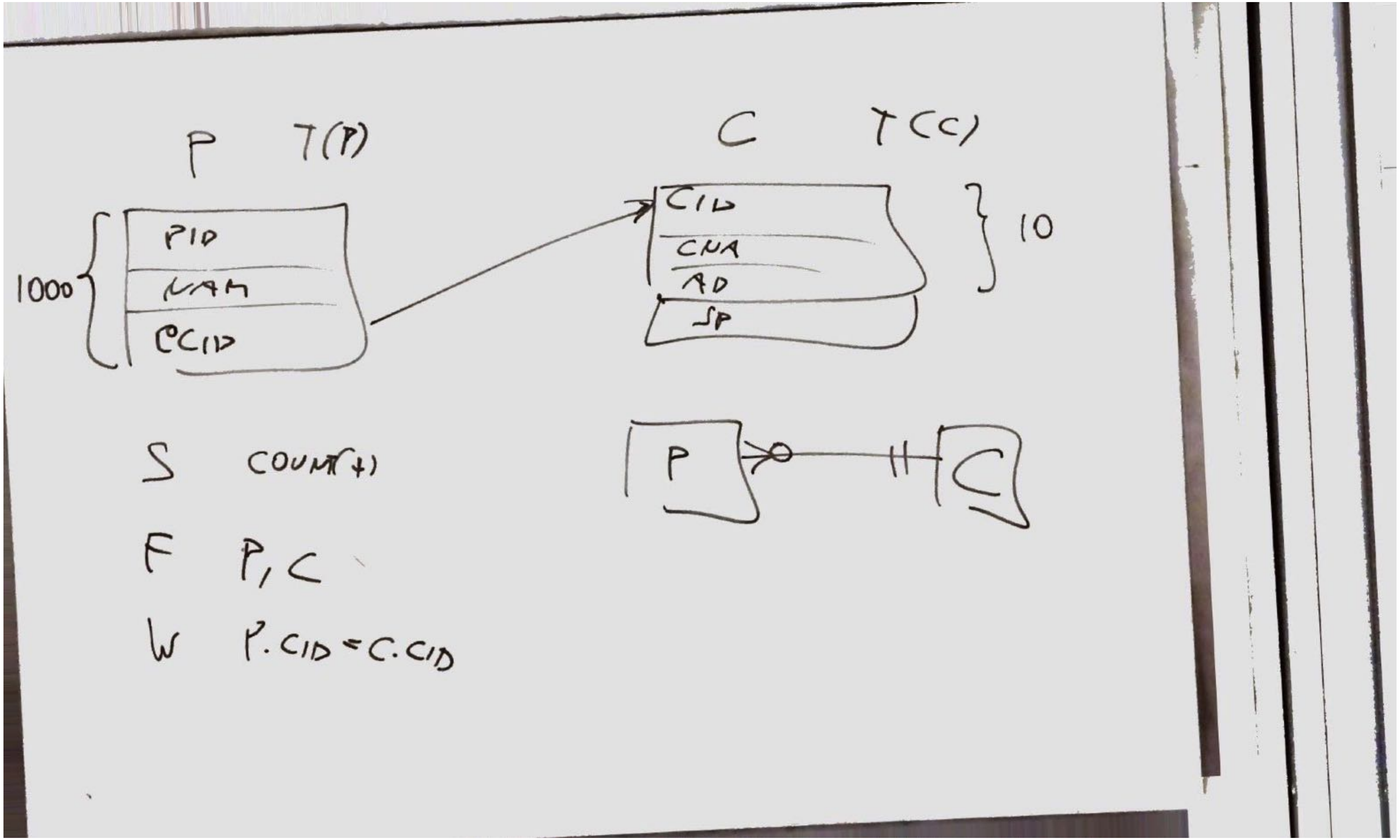
$$\begin{array}{l}50 \times 5 = 250\\ 50 \times 5 = \underline{250}\\ \quad\quad\quad\;\; 500\end{array}$$

(m, m) m = m
(f, f) f = f
(f, m) f = m
(m, f)

Whiteboard example

- Assume the table product has 1000 entries and the table company has 100 entries, cid is the PK in company, and a FK in product.
- How large is the result of following query?

```
SELECT *
FROM   Product P, Company C
WHERE  P.cid = C.cid
```



S COUNT(*)

F P, C

W P.CID = C.CID