Becoming a better developer with explain

Understanding Postgres Query planner

Louise Grandjonc

About me

Louise Grandjonc (louise@ulule.com)
Lead developer at Ulule (www.ulule.com)
Python / Django developer - Postgres enthusiast
@louisemeta on twitter

Today's agenda

1. Can understanding the query plan help me?
2. What if I'm using an ORM?
3. So what does my database do when I query stuff?

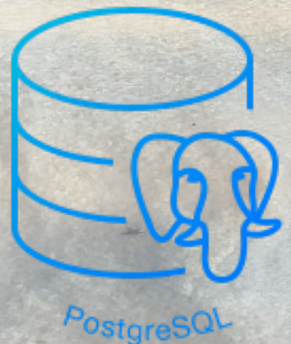
Can understanding the query plan help me?

The mystery of querying stuff

You

Query

```
SELECT …
FROM …
```



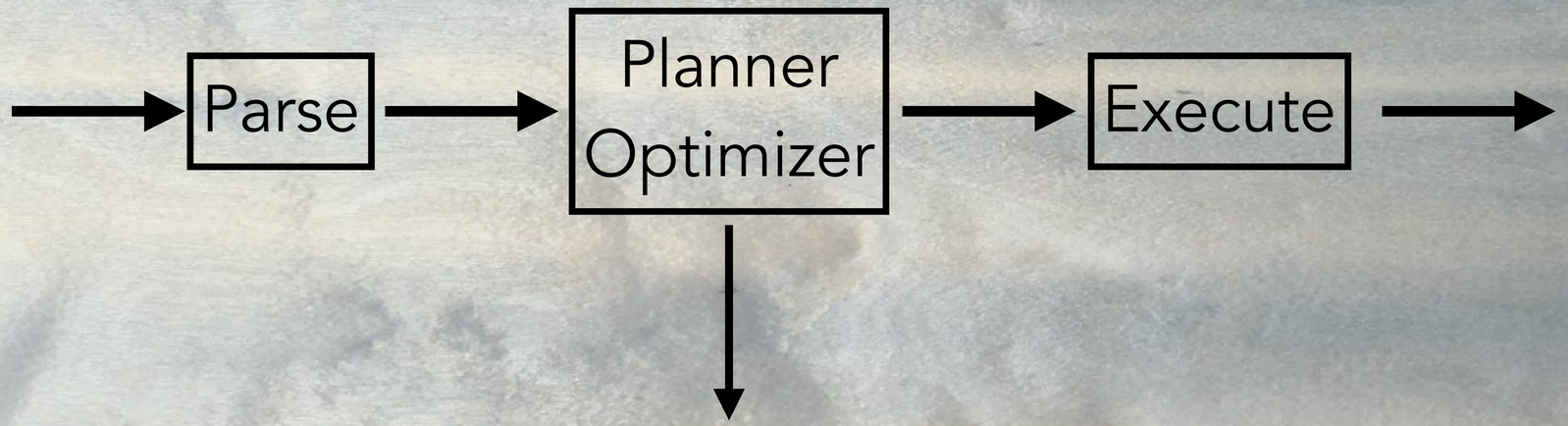
Result

id	name
1	Louise
2	Alfred
…	…

Zooming into it

SQL Query

```
SELECT ...
FROM ...
```



Result

id	name
1	Louise
2	Alfred
...	...

- Generates execution plans for a query
- Calculates the cost of each plan.
- The best one is used to execute your query

So what can I learn from the query plan

- Understand why is your filter / join / order slow.
- Know if the new index you just added is used.
- Stop guessing which index can be useful

In conclusion: You will understand why your query is slow

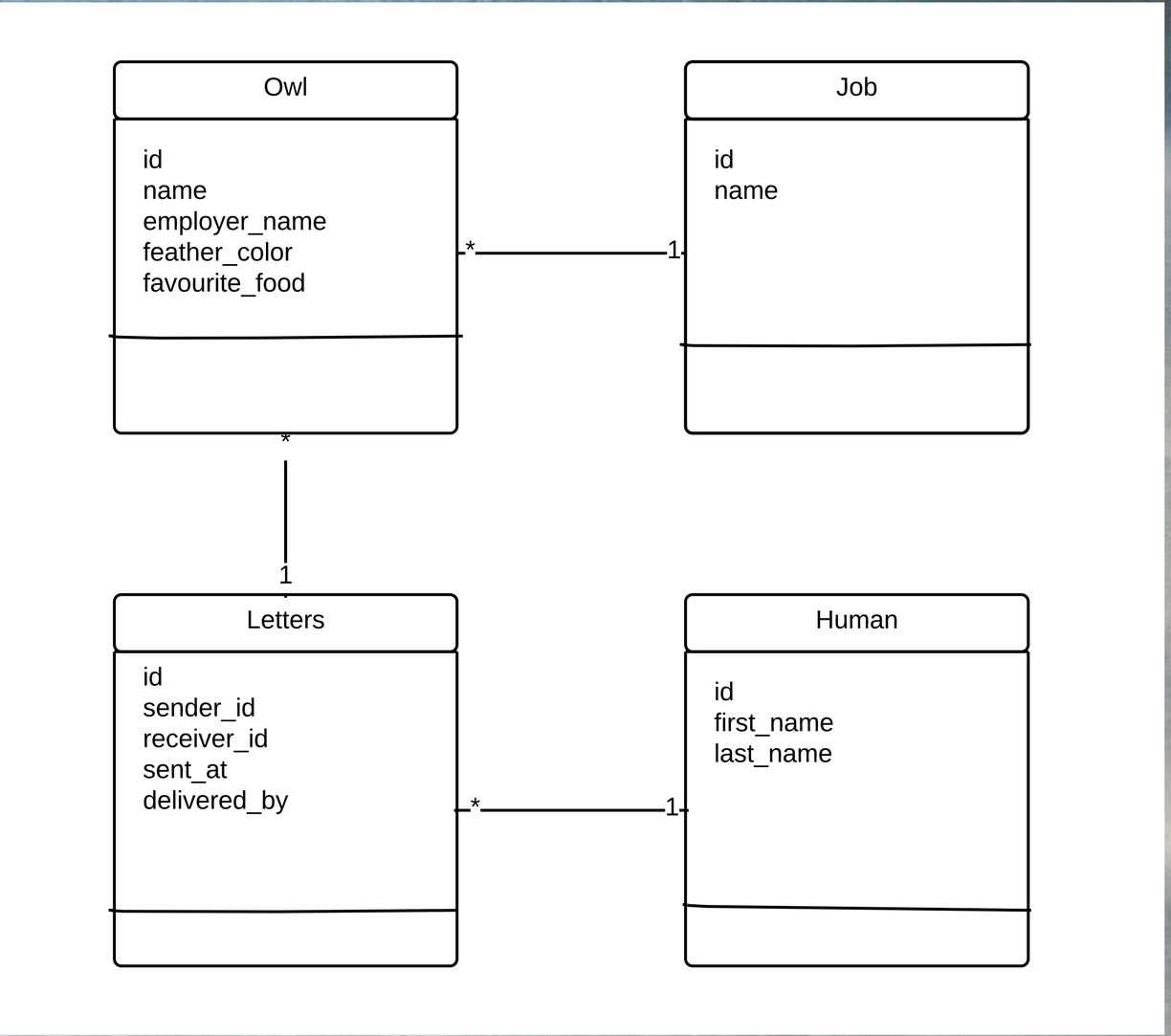
What if I'm using an ORM ?

But why can’t we trust it ?

1. The ORM executes queries that you might not expect
2. Your queries might not be optimised and you won’t know about it

The story of the owls

10 002 owls
10 000 humans
411861 letters



Loops (with django)

```
owls = Owl.objects.filter(employer_name='Hogwarts')
for owl in owls:
    print(owl.job) # 1 query per loop iteration
```

```
SELECT id, name, employer_name, favourite_food, job_id,
feather_color FROM owl WHERE employer_name = 'Hogwarts'
SELECT id, name FROM job WHERE id = 1
SELECT id, name FROM job WHERE id = 1
SELECT id, name FROM job WHERE id = 1
SELECT id, name FROM job WHERE id = 1
SELECT id, name FROM job WHERE id = 1
SELECT id, name FROM job WHERE id = 1
SELECT id, name FROM job WHERE id = 1
SELECT id, name FROM job WHERE id = 1
…
```

Awesome right ?

Loops (with django)

```
Owl.objects.filter(employer_name='Ulule')
    .select_related('job')
```

```
SELECT … FROM "owl" LEFT OUTER JOIN "job" ON ("owl"."job_id" =
"job"."id")
WHERE "owl"."employer_name" = 'Ulule'
```

```
Owl.objects.filter(employer_name='Ulule')
.prefetch_related('job')
```

```
SELECT … FROM "owl" WHERE "owl"."employer_name" = 'Ulule'
SELECT … FROM "job" WHERE "job"."id" IN (2)
```

Where are my logs?

Terminal command

```
$ psql -U user -d your_database_name
```

psql interface

```
owl_conference=# show log_directory ;
 log_directory
---------------
 pg_log

owl_conference=# show data_directory ;
     data_directory
--------------------------
 /usr/local/var/postgres

owl_conference=# show log_filename ;
      log_filename
--------------------------
 postgresql-%Y-%m-%d.log
```

Having good looking logs
(and logging everything like a crazy owl)

```
owl_conference=# SHOW config_file;
                config_file
-----------------------------------------
 /usr/local/var/postgres/postgresql.conf
(1 row)
```

In your postgresql.conf

```
log_filename = 'postgresql-%Y-%m-%d.log'
log_statement = 'all'
logging_collector = on
log_min_duration_statement = 0
```

Finding dirty queries
pg_stat_statements

In your psql

```
CREATE EXTENSION pg_stat_statements;
```

The module must be added to your shared_preload_libraries.

You have to change your postgresql.conf and restart.

```
shared_preload_libraries = 'pg_stat_statements'
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

Finding the painful queries

```
SELECT total_time, min_time, max_time, mean_time,
calls, query
FROM pg_stat_statements
ORDER BY mean_time DESC
LIMIT 100;


-[ RECORD 6 ]-------------------------------------------------
total_time | 519.491
min_time   | 519.491
max_time   | 519.491
mean_time  | 519.491
calls      | 1
query      | SELECT COUNT(*) FROM letters;
```

What is my database doing when I query?

Let’s talk about EXPLAIN !

What is EXPLAIN

```
EXPLAIN (ANALYZE) my super query;
```

Gives you the execution plan chosen by the query planner that your database will use to execute your SQL statement

Using ANALYZE will actually execute your query! (Don’t worry, you can ROLLBACK)

```
BEGIN;
EXPLAIN ANALYZE UPDATE owl SET … WHERE …;
ROLLBACK;
```

So, what does it took like ?

```
EXPLAIN ANALYZE SELECT * FROM owl WHERE
employer_name='Ulule';

 QUERY PLAN
-----------------------------------
Seq Scan on owl  (cost=0.00..205.01 rows=1 width=35)
           (actual time=1.945..1.946 rows=1 loops=1)
   Filter: ((employer_name)::text = 'Ulule'::text)
   Rows Removed by Filter: 10001
 Planning time: 0.080 ms
 Execution time: 1.965 ms
(5 rows)
```

Let’s go step by step ! .. 1

Costs

```
(cost=0.00..205.01 rows=1 width=35)
```

Cost of retrieving first row

Cost of retrieving all rows

Number of rows returned

Average width of a row (in bytes)

If you use ANALYZE

```
(actual time=1.945..1.946 rows=1 loops=1)
```

Number of time your seq scan (index scan etc.) was executed

Sequential Scan

```
Seq Scan on owl  ...
Filter: ((employer_name)::text = 'Ulule'::text)
   Rows Removed by Filter: 10001
```

- Scan the entire database table.
- Retrieves the rows matching your WHERE.

It can be expensive !

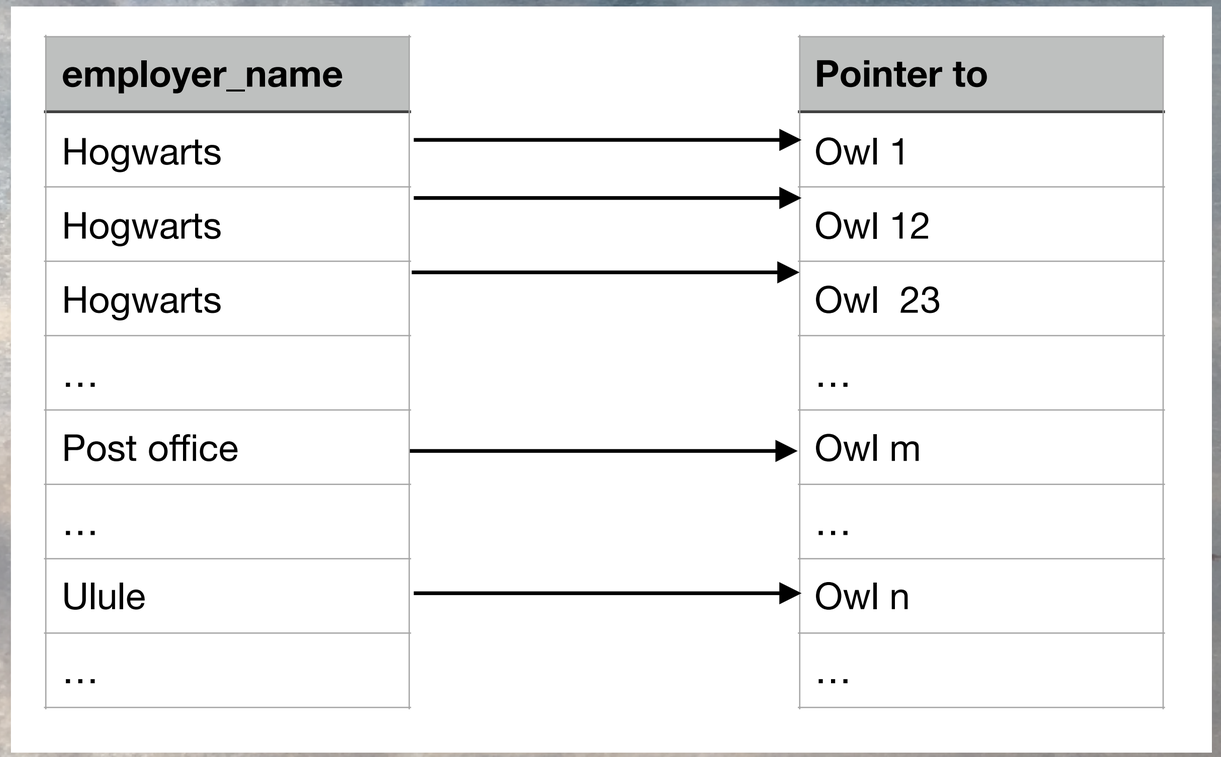
Would an index make this query faster ?

What is an index then?

```
CREATE INDEX ON owl (employer_name);
```

In an encyclopaedia, if you want to read about owls, you don't read the entire book, you go to the index first!

A database index contains the column value and pointers to the row that has this value.



Index scan

```
Index Scan using owl_employer_name_idx on owl
(cost=0.29..8.30 rows=1 width=35) (actual
time=0.055..0.056 rows=1 loops=1)
   Index Cond: ((employer_name)::text =
'Ulule'::text)
 Planning time: 0.191 ms
 Execution time: 0.109 ms
```

The index is visited row by row in order to retrieve the data corresponding to your clause.

Index scan or sequential scan?

With an index and a really common value !
7000/10 000 owls work at the post office

```
EXPLAIN SELECT * FROM owl
WHERE employer_name = 'post office';

QUERY PLAN
-----------------------------------------------
Seq Scan on owl  (cost=0.00..205.03 rows=7001 width=35)
   Filter: ((employer_name)::text = 'post office'::text)
```

Why is it using a sequential scan?

An index scan uses the order of the index, the head has to move between rows.

Moving the read head of the database is 1000 times slower than reading the next physical block.

Conclusion: For common values it’s quicker to read all data from the table in physical order

By the way… Retrieving 7000 rows might not be a great idea :).

Bitmap Heap Scan

With an index and a common value
2000 owls work at Hogwarts

```
EXPLAIN ANALYZE SELECT * FROM owl WHERE owl.employer_name = 'Hogwarts';

QUERY PLAN
-----------------------------------------------
Bitmap Heap Scan on owl  (cost=23.50..128.50 rows=2000 width=35)
(actual time=1.524..4.081 rows=2000 loops=1)
   Recheck Cond: ((employer_name)::text = 'Hogwarts'::text)
   Heap Blocks: exact=79
   ->  Bitmap Index Scan on owl_employer_name_idx1  (cost=0.00..23.00
rows=2000 width=0) (actual time=1.465..1.465 rows=2000 loops=1)
         Index Cond: ((employer_name)::text = 'Hogwarts'::text)
 Planning time: 15.642 ms
 Execution time: 5.309 ms
(7 rows)
```

Bitmap Heap Scan

- The tuple-pointer from index are ordered by physical memory
- Go through the map.

Limit physical jumps between rows.

Recheck condition ? If the bitmap is too big

- The bitmap contains the pages where there are rows
- Goes through the pages, rechecks the condition

So we have 3 types of scan

1. Sequential scan
2. Index scan
3. Bitmap heap scan

And now let’s join stuff !

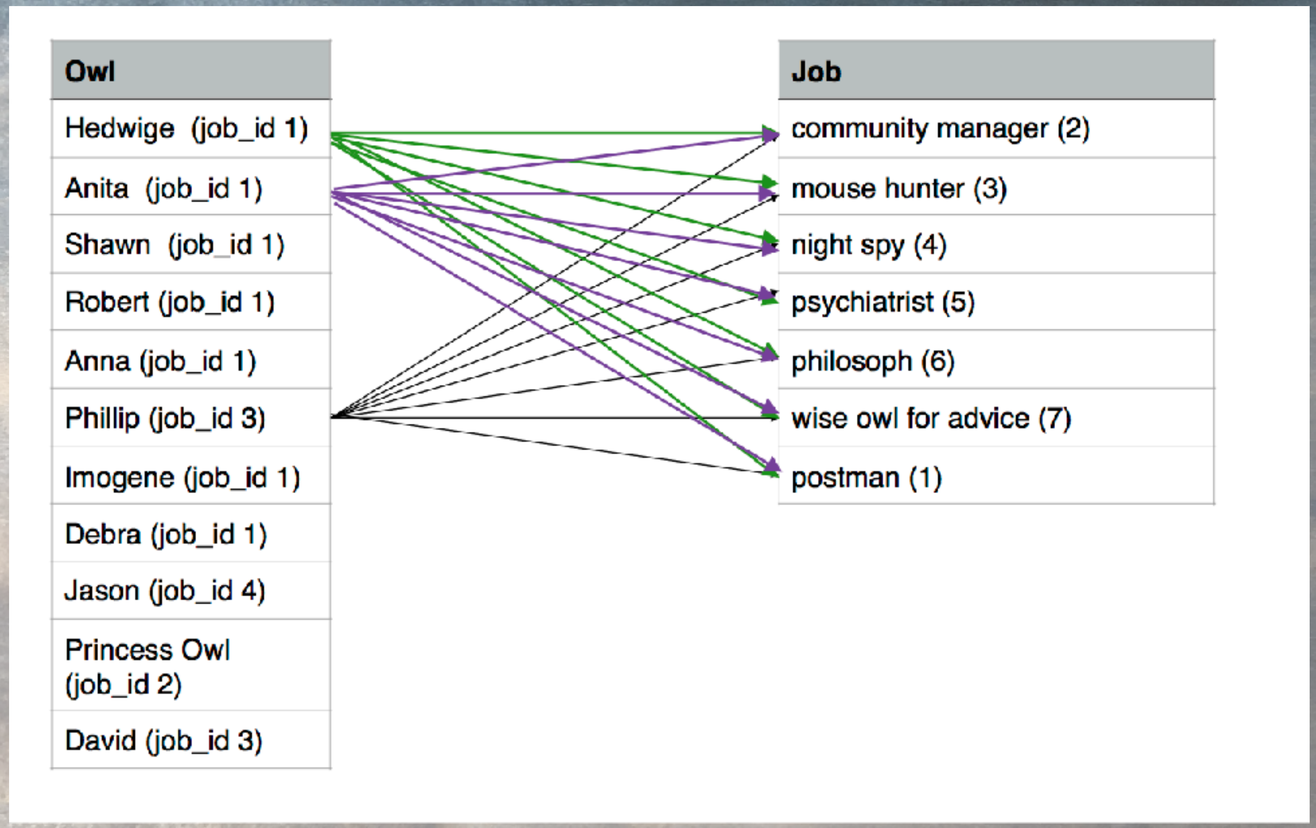
And now let's join !

Nested loops

```
EXPLAIN ANALYZE SELECT * FROM owl JOIN job ON (job.id = owl.job_id)
WHERE job.id=1;

QUERY PLAN
------------------------------------------------------------------------
 Nested Loop  (cost=0.00..296.14 rows=9003 width=56)
              (actual time=0.093..4.081 rows=9001 loops=1)
   ->  Seq Scan on job  (cost=0.00..1.09 rows=1 width=21) (actual
time=0.064..0.064 rows=1 loops=1)
         Filter: (id = 1)
         Rows Removed by Filter: 6
   ->  Seq Scan on owl  (cost=0.00..205.03 rows=9003 width=35)
(actual time=0.015..2.658 rows=9001 loops=1)
         Filter: (job_id = 1)
         Rows Removed by Filter: 1001
 Planning time: 0.188 ms
 Execution time: 4.757 ms
```

Nested loops
Python version

```
jobs = Job.objects.all()
owls = Owl.objects.all()

for owl in owls:
    for job in jobs:
        if owl.job_id == job.id:
            owl.job = job
            break
```



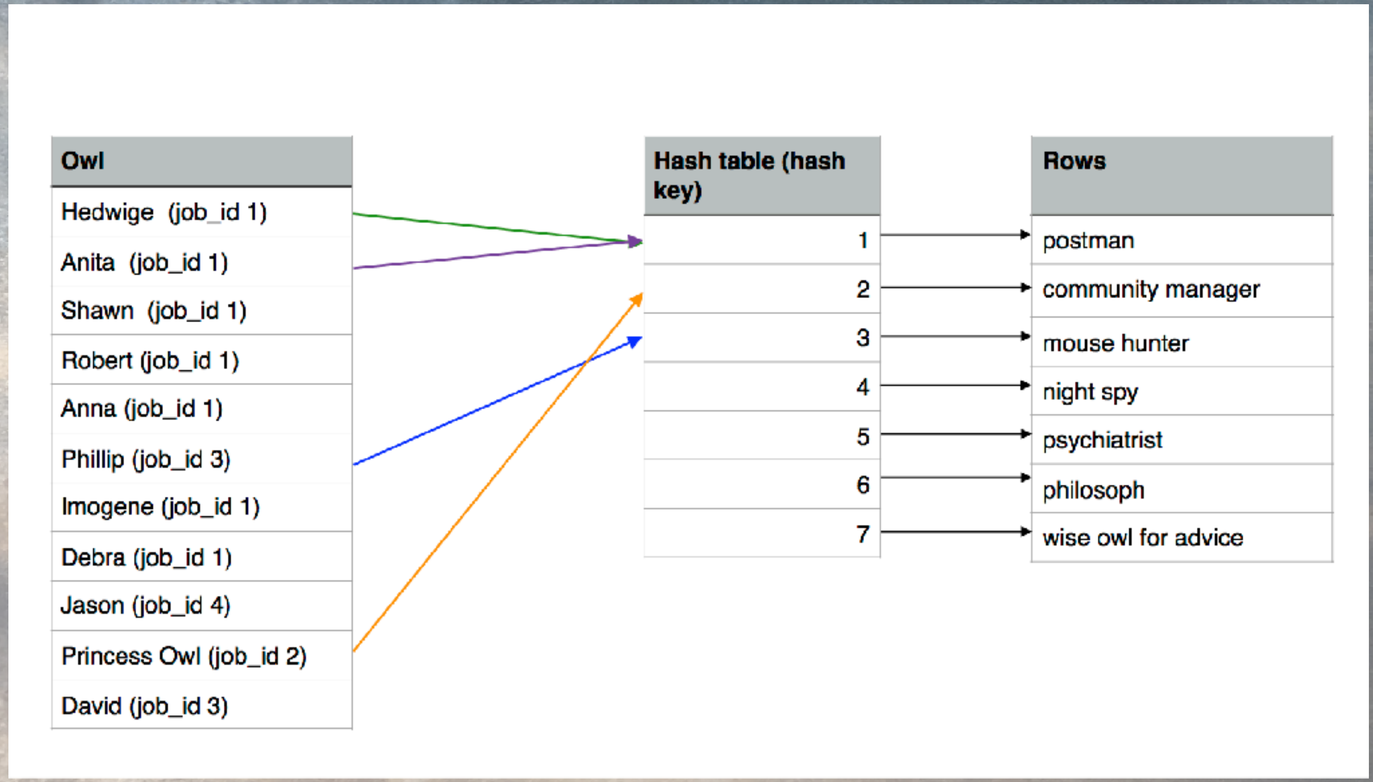
- Used for little tables
- Complexity of O(n*m)

Hash Join

```
EXPLAIN ANALYZE SELECT * FROM owl JOIN job ON (job.id = owl.job_id) WHERE job.id > 1;

QUERY PLAN
------------------------------------------------------------------------------
Hash Join  (cost=1.17..318.70 rows=10001 width=56) (actual time=0.058..3.830 rows=1000 loops=1)
   Hash Cond: (owl.job_id = job.id)
   ->  Seq Scan on owl  (cost=0.00..180.01 rows=10001 width=35) (actual time=0.039..2.170
rows=10002 loops=1)
   ->  Hash  (cost=1.09..1.09 rows=7 width=21) (actual time=0.010..0.010 rows=6 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Seq Scan on job  (cost=0.00..1.09 rows=7 width=21) (actual time=0.007..0.009 rows=6
loops=1)
               Filter: (id > 1)
               Rows Removed by Filter: 1
 Planning time: 2.327 ms
 Execution time: 3.905 ms
(10 rows)
```

Hash Join
Python version

```
jobs = Job.objects.all()

jobs_dict = {}
for job in jobs:
    jobs_dict[job.id] = job

owls = Owl.objects.all()
for owl in owls:
    owl.job = jobs_dict[owl.job_id]
```

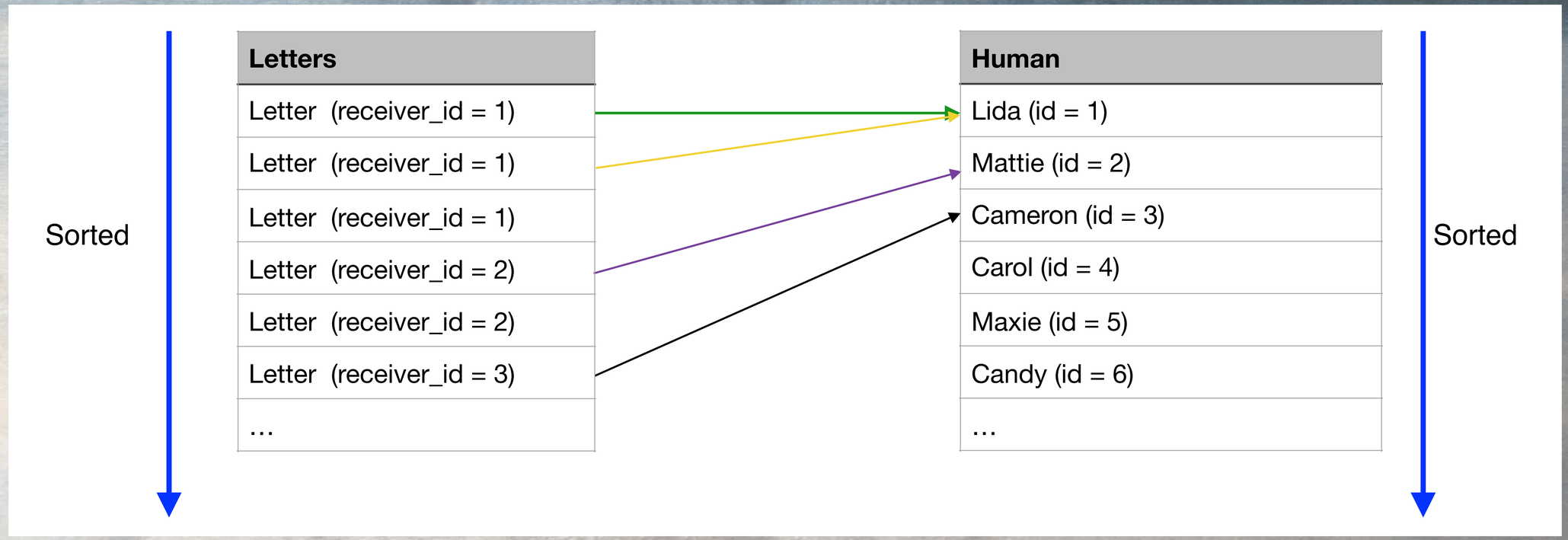


- Used for small tables
- the hash table has to fit in memory (You wouldn't do a Python dictionary with 1M rows ;))
- If the table is really small, a nested loop is used because of the complexity of creating a hash table

Merge Join

```
EXPLAIN ANALYZE SELECT * FROM letters JOIN human ON (receiver_id = human.id) ORDER BY
letters.receiver_id;

QUERY PLAN
-----------------------------------------------------------------
Merge Join  (cost=55300.94..62558.10 rows=411861 width=51)
            (actual time=264.207..502.215 rows=411861 loops=1)
   Merge Cond: (human.id = letters.receiver_id)
   ->  Sort  (cost=894.39..919.39 rows=10000 width=23) (actual time=3.191..4.413 rows=10000 loops=1)
         Sort Key: human.id
         Sort Method: quicksort  Memory: 1166kB
         ->  Seq Scan on human  (cost=0.00..230.00 rows=10000 width=23) (actual time=0.067..1.131
rows=10000 loops=1)
   ->  Materialize  (cost=54406.35..56465.66 rows=411861 width=24) (actual time=261.009..414.465
rows=411861 loops=1)
         ->  Sort  (cost=54406.35..55436.00 rows=411861 width=24) (actual time=261.008..369.777
rows=411861 loops=1)
               Sort Key: letters.receiver_id
               Sort Method: external merge  Disk: 15232kB
               ->  Seq Scan on letters  (cost=0.00..7547.61 rows=411861 width=24) (actual
time=0.010..37.253 rows=411861 loops=1)
 Planning time: 0.205 ms
 Execution time: 537.119 ms
(13 rows)
```

Merge Join - 2



Used for big tables, an index can be used to avoid sorting

So we have 3 types of joins

1. Nested loop
2. Hash join
3. Merge join

And now, ORDER BY

And now let's order stuff…

```
EXPLAIN ANALYZE SELECT * FROM human ORDER BY last_name;

QUERY PLAN
-----------------------------------------------------------------
Sort  (cost=894.39..919.39 rows=10000 width=23)
      (actual time=163.228..164.211 rows=10000 loops=1)
   Sort Key: last_name
   Sort Method: quicksort  Memory: 1166kB
   ->  Seq Scan on human  (cost=0.00..230.00 rows=10000 width=23)
(actual time=14.341..17.593 rows=10000 loops=1)
 Planning time: 0.189 ms
 Execution time: 164.702 ms
(6 rows)
```

Everything is sorted in the memory
(which is why it can be costly in terms of memory)

ORDER BY LIMIT

```
EXPLAIN ANALYZE SELECT * FROM human
ORDER BY last_name LIMIT 3;

 QUERY PLAN
-----------------------------------------------------------------
Limit  (cost=446.10..446.12 rows=10 width=23)
       (actual time=11.942..11.944 rows=3 loops=1)
   ->  Sort  (cost=446.10..471.10 rows=10000 width=23)
             (actual time=11.942..11.942 rows=3 loops=1)
         Sort Key: last_name
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Seq Scan on human  (cost=0.00..230.00 rows=10000
width=23) (actual time=0.074..0.947 rows=10000 loops=1)
 Planning time: 0.074 ms
 Execution time: 11.966 ms
(7 rows)
```

Like with quicksort, all the data has to be sorted… Why is the memory taken so much smaller?

Top-N heap sort

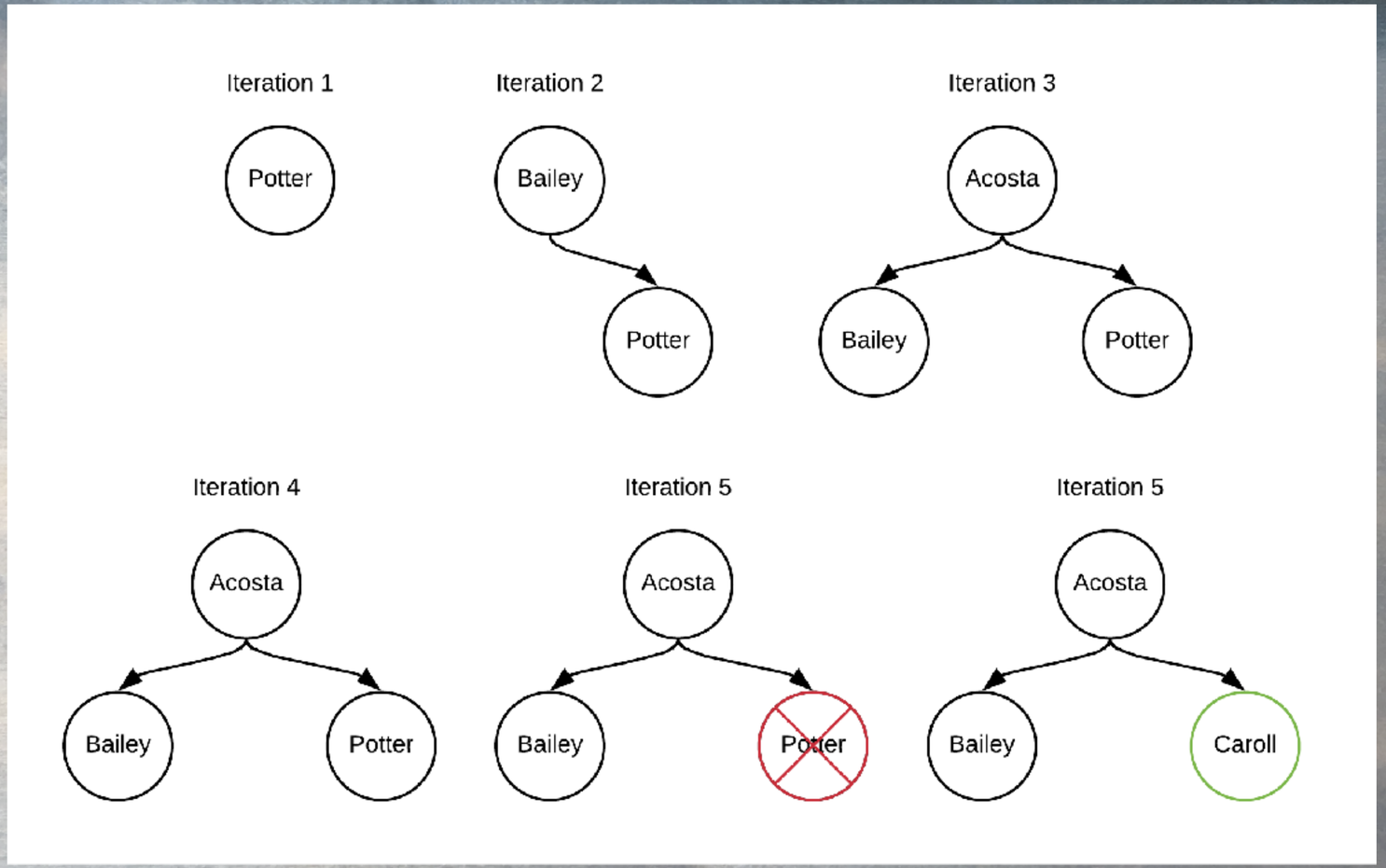
- A heap (sort of tree) is used with a limited size
- For each row
  - If heap not full: add row in heap
  - Else
    - If value smaller than current values (for ASC): insert row in heap, pop last
    - Else pass

Top-N heap sort

Example LIMIT 3

Human

id	last_name
1	Potter
2	Bailey
3	Acosta
4	Weasley
5	Caroll
...	...



Ordering with an index

```
CREATE INDEX ON human (last_name);
```

```
EXPLAIN ANALYZE SELECT * FROM human ORDER BY last_name LIMIT 5;

QUERY PLAN
-----------------------------------------------------------
Limit  (cost=0.29..0.70 rows=5 width=23)
       (actual time=0.606..0.611 rows=5 loops=1)
   ->  Index Scan using human_last_name_idx on human
(cost=0.29..834.27 rows=10000 width=23)
(actual time=0.605..0.610 rows=5 loops=1)
 Planning time: 0.860 ms
 Execution time: 0.808 ms
(4 rows)
```

Simply uses index order

Be careful when you ORDER BY !

1. Sorting with sort key without limit or index can be heavy in term of memory !
2. You might need an index, only EXPLAIN will tell you

Let's get Sirius !

An example a bit more complex

The ministry of magic wants to identify suspect humans.
To do that they ask their DBM (Database Magician) for

- The list of the letters sent by Voldemort
- For each we want the id and date of the reply (if exists)
- All of this with the first and last name of the receiver

Expected result

first_name	last_name	receiver_id	letter_id	sent_at	answer_id	answer_sent_at
Leticia	O'Takatakaer	812	577732	2010-03-05 20:18:00		
Almeda	Steinkingjacer	899	577735	2010-03-09 11:44:00		
Sherita	Butchchterjac	1444	577744	2010-03-18 09:11:00		
Devon	Jactakadurmc	1812	577760	2010-04-02 14:38:00		
Mariah	Mctakachenmc	2000	577733	2010-03-06 18:00:00		
Lupita	Duro'Otosan	2332	577751	2010-03-24 18:04:00		
Chloe	Royo'Erfür	2464	577737	2010-03-10 14:46:00		
Agustin	Erillsancht	2668	577752	2010-03-25 22:57:00		
Hiram	Ibnchensteinill	2689	577746	2010-03-19 13:55:00		
Jayme	Erersteinvur	2856	577766	2010-04-08 23:03:00	578834	2010-6-16 6:26:36

The python version

```
letters_from_voldemort =
Letters.objects.filter(sender_id=3267).select_related('receiver').order_by('sent_at')
letters_to_voldemort = Letters.objects.filter(receiver_id=3267).order_by('sent_at')
data = []
for letter in letters_from_voldemort:
   for answer in letters_to_voldemort:
        if letter.receiver_id == answer.sender_id and letter.sent_at < answer.sent_at:
            answer_found = True
            data.append([letter.receiver.first_name, letter.receiver.last_name,
letter.receiver.id, letter.id, letter.sent_at, answer.id, answer.sent_at])
            break
   else:
       data.append([letter.receiver.first_name, letter.receiver.last_name,
letter.receiver.id, letter.id, letter.sent_at])
```

Takes about 1540 ms

Why not have fun with SQL ?

```
SELECT human.first_name, human.last_name, receiver_id, letter_id,
       sent_at, answer_id, answer_sent_at
FROM (
  SELECT
    id as letter_id, receiver_id, sent_at, sender_id
  FROM letters
  WHERE
        sender_id=3267
  ORDER BY sent_at
) l1 LEFT JOIN LATERAL (
  SELECT
    id as answer_id,
    sent_at as answer_sent_at
  FROM letters
  WHERE
    sender_id = l1.receiver_id
    AND sent_at > l1.sent_at
    AND receiver_id = l1.sender_id
  LIMIT 1
) l2 ON true JOIN human ON (human.id=receiver_id)
```

Oh well, it takes 48 seconds…

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=8579.33..434310.73 rows=40 width=47) (actual time=99.106..48459.498 rows=1067
loops=1)
   ->  Hash Join  (cost=8579.33..8847.23 rows=40 width=39) (actual time=53.903..60.985 rows=1067 loops=1)
         Hash Cond: (human.id = l1.receiver_id)
         ->  Seq Scan on human  (cost=0.00..230.00 rows=10000 width=23) (actual time=0.060..1.303 rows=10000
loops=1)
         ->  Hash  (cost=8578.83..8578.83 rows=40 width=20) (actual time=53.828..53.828 rows=1067 loops=1)
               Buckets: 2048 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 71kB
               ->  Subquery Scan on l1  (cost=8578.33..8578.83 rows=40 width=20) (actual time=53.377..53.580
rows=1067 loops=1)
                     ->  Sort  (cost=8578.33..8578.43 rows=40 width=20) (actual time=53.376..53.439 rows=1067
loops=1)
                           Sort Key: letters.sent_at
                           Sort Method: quicksort  Memory: 132kB
                           ->  Seq Scan on letters  (cost=0.00..8577.26 rows=40 width=20) (actual
time=0.939..53.127 rows=1067 loops=1)
                                 Filter: (sender_id = 3267)
                                 Rows Removed by Filter: 410794
   ->  Limit  (cost=0.00..10636.57 rows=1 width=12) (actual time=45.356..45.356 rows=0 loops=1067)
         ->  Seq Scan on letters letters_1  (cost=0.00..10636.57 rows=1 width=12) (actual time=45.346..45.346
rows=0 loops=1067)
               Filter: ((sent_at > l1.sent_at) AND (sender_id = l1.receiver_id) AND (receiver_id = l1.sender_id))
               Rows Removed by Filter: 410896
 Planning time: 0.859 ms
 Execution time: 48460.225 ms
(19 rows)
```

What is my problem ?

```
     ->  Sort  (cost=8578.33..8578.43 rows=40 width=20) (actual
time=53.376..53.439 rows=1067 loops=1)
           Sort Key: letters.sent_at
           Sort Method: quicksort  Memory: 132kB
           ->  Seq Scan on letters  (cost=0.00..8577.26 rows=40
width=20) (actual time=0.939..53.127 rows=1067 loops=1)
                 Filter: (sender_id = 3267)
                 Rows Removed by Filter: 410794
```

```
->  Limit  (cost=0.00..10636.57 rows=1 width=12) (actual
time=45.356..45.356 rows=0 loops=1067)
        ->  Seq Scan on letters letters_1  (cost=0.00..10636.57
rows=1 width=12) (actual time=45.346..45.346 rows=0
loops=1067)
              Filter: ((sent_at > l1.sent_at) AND (sender_id =
l1.receiver_id) AND (receiver_id = l1.sender_id))
              Rows Removed by Filter: 410896
```

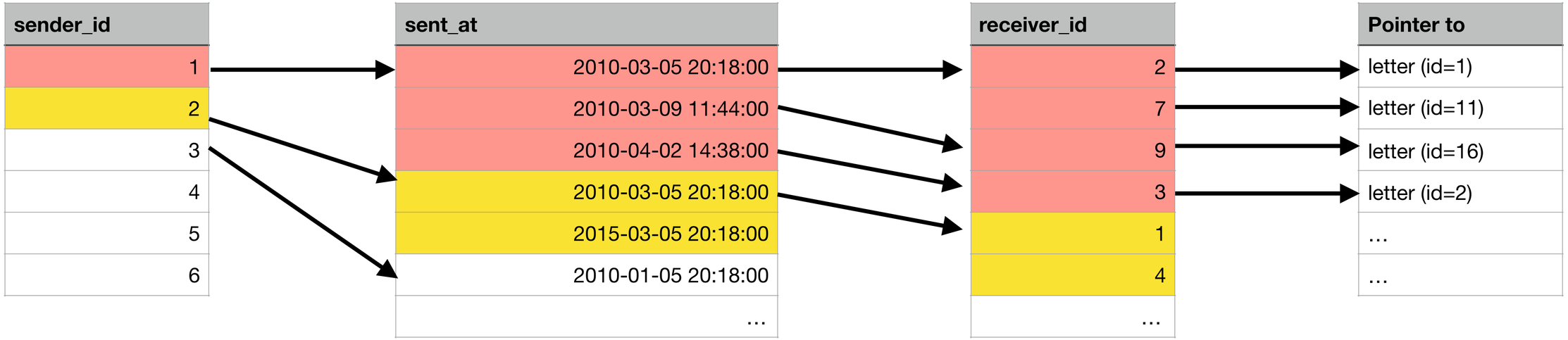
The query planner is using a sequential scan to filter almost our entire table…

So clearly we are missing some indexes on the columns receiver_id, sender_id and sent_at.

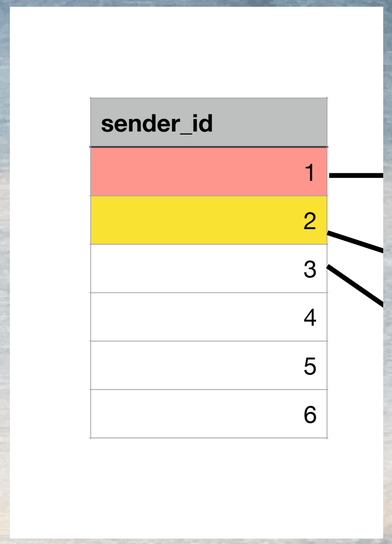
Let’s create the following index

```
CREATE INDEX ON letters (sender_id, sent_at, receiver_id);
```

Quick reminder on multi column indexes



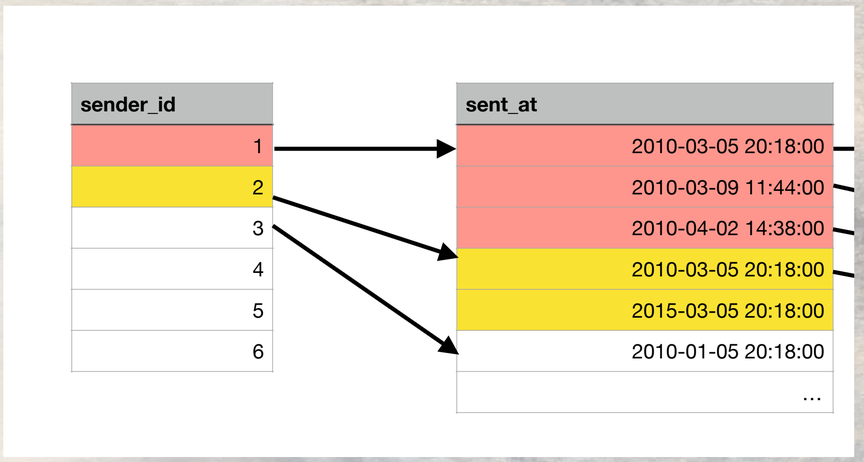
Multi-columns indexes



The first column of the index is ordered

The index will be used for

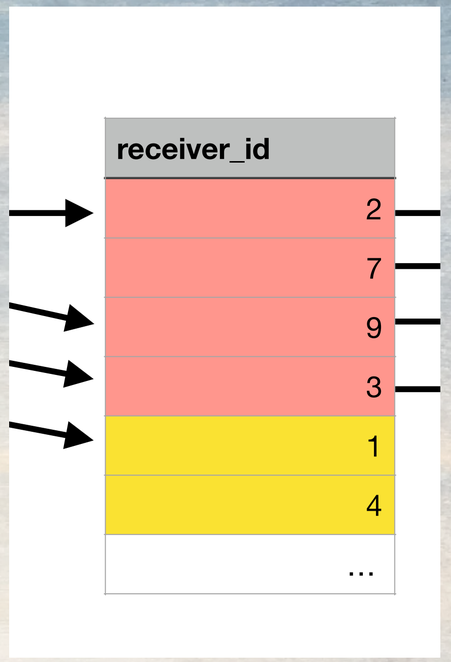
```
SELECT … FROM letters WHERE sender_id=…;
```



The same is true for

```
SELECT … FROM letters WHERE sender_id=…
AND sent_at = …;
```

The order of the columns matters !



The index won't be used for

```
SELECT … FROM letters WHERE receiver_id=…;
```

In our previous query we had

```
SELECT … FROM letters WHERE sender_id=3267 ORDER BY sent_at
```

And then in the LATERAL JOIN

```
SELECT … FROM letters WHERE sender_id = l1.receiver_i
AND sent_at > l1.sent_at
AND receiver_id = l1.sender_id
```

The index (sender_id, sent_at, receiver_id) works fine

Better no ? 180 times faster than Python

```
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=154.76..767.34 rows=40 width=47) (actual time=1.092..7.694 rows=1067 loops=1)
   ->  Hash Join  (cost=154.34..422.24 rows=40 width=39) (actual time=1.080..3.471 rows=1067 loops=1)
         Hash Cond: (human.id = l1.receiver_id)
         ->  Seq Scan on human  (cost=0.00..230.00 rows=10000 width=23) (actual time=0.076..0.797 rows=10000 loops=1)
         ->  Hash  (cost=153.84..153.84 rows=40 width=20) (actual time=0.992..0.992 rows=1067 loops=1)
               Buckets: 2048 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 71kB
               ->  Subquery Scan on l1  (cost=153.34..153.84 rows=40 width=20) (actual time=0.520..0.813 rows=1067 loops=1)
                     ->  Sort  (cost=153.34..153.44 rows=40 width=20) (actual time=0.520..0.630 rows=1067 loops=1)
                           Sort Key: letters.sent_at
                           Sort Method: quicksort  Memory: 132kB
                           ->  Bitmap Heap Scan on letters  (cost=4.73..152.27 rows=40 width=20) (actual time=0.089..0.304
rows=1067 loops=1)
                                 Recheck Cond: (sender_id = 3267)
                                 Heap Blocks: exact=74
                                 ->  Bitmap Index Scan on letters_sender_id_sent_at_receiver_id_idx  (cost=0.00..4.72
rows=40 width=0) (actual time=0.079..0.079 rows=1067 loops=1)
                                       Index Cond: (sender_id = 3267)
   ->  Limit  (cost=0.42..8.61 rows=1 width=12) (actual time=0.004..0.004 rows=0 loops=1067)
         ->  Index Scan using letters_sender_id_sent_at_receiver_id_idx on letters letters_1  (cost=0.42..8.61 rows=1
width=12) (actual time=0.003..0.003 rows=0 loops=1067)
               Index Cond: ((sender_id = l1.receiver_id) AND (sent_at > l1.sent_at) AND (receiver_id = l1.sender_id))
 Planning time: 0.565 ms
 Execution time: 7.804 ms
(20 rows)
```

Better no ?

```
->  Bitmap Heap Scan on letters  (cost=4.73..152.27 rows=40 width=20) (actual time=0.089..0.304 rows=1067 loops=1)
      Recheck Cond: (sender_id = 3267)
      Heap Blocks: exact=74
      ->  Bitmap Index Scan on letters_sender_id_sent_at_receiver_id_idx  (cost=0.00..4.72 rows=40 width=0) (actual
time=0.079..0.079 rows=1067 loops=1)
            Index Cond: (sender_id = 3267)
```

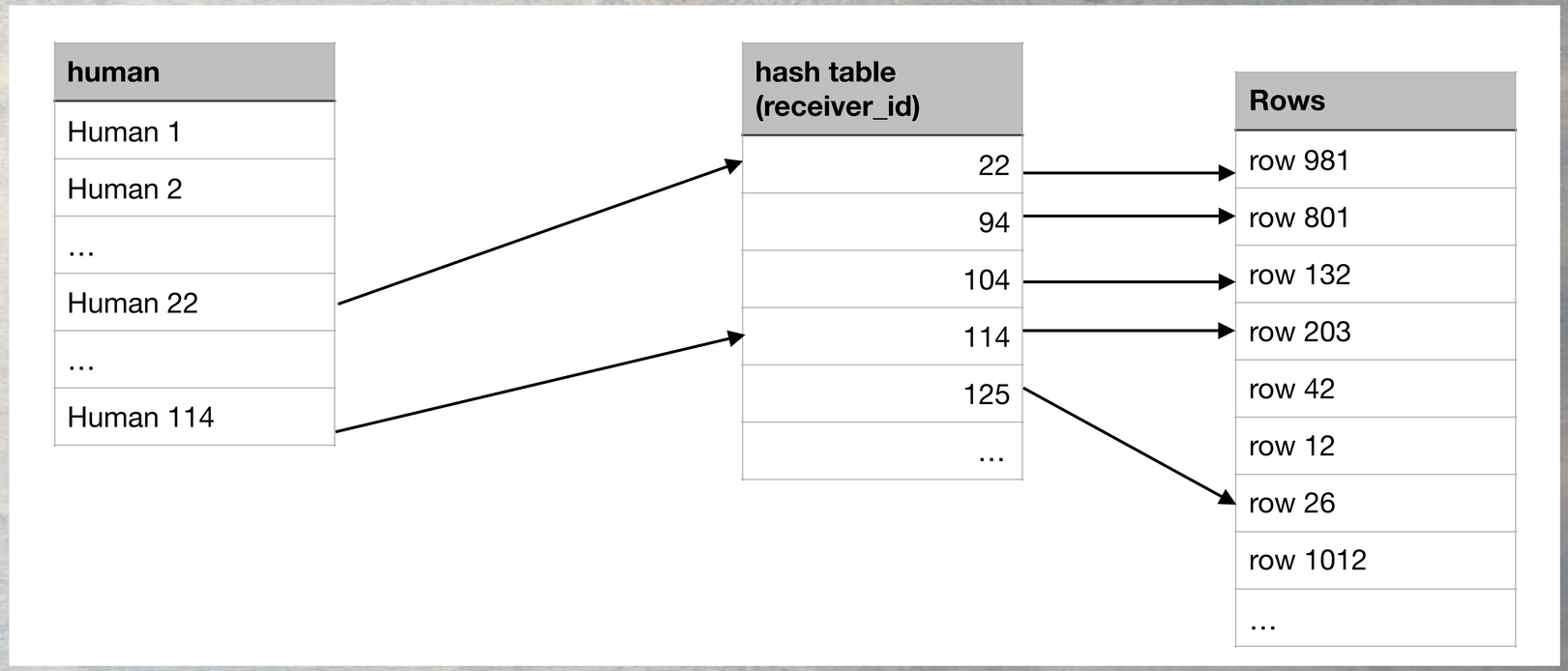
```
->  Limit  (cost=0.42..8.61 rows=1 width=12) (actual time=0.004..0.004 rows=0 loops=1067)
        ->  Index Scan using letters_sender_id_sent_at_receiver_id_idx on letters letters_1  (cost=0.42..8.61 rows=1 width=12)
(actual time=0.003..0.003 rows=0 loops=1067)
              Index Cond: ((sender_id = l1.receiver_id) AND (sent_at > l1.sent_at) AND (receiver_id = l1.sender_id))
```

The index is used where there were sequential scans.

Focusing on this has join…

```
->  Hash Join  (cost=154.34..422.24 rows=40 width=39) (actual time=1.080..3.471 rows=1067 loops=1)
      Hash Cond: (human.id = l1.receiver_id)
      ->  Seq Scan on human  (cost=0.00..230.00 rows=10000 width=23) (actual time=0.076..0.797 rows=10000 loops=1)
      ->  Hash  (cost=153.84..153.84 rows=40 width=20) (actual time=0.992..0.992 rows=1067 loops=1)
            Buckets: 2048 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 71kB
            ->  Subquery Scan on l1  (cost=153.34..153.84 rows=40 width=20) (actual time=0.520..0.813 rows=1067 loops=1)
```

A hash is built from the subquery (1067 rows)
Human table having 10 000 rows



Let’s use a pagination…

```
SELECT human.first_name, human.last_name, receiver_id, letter_id,
       sent_at, answer_id, answer_sent_at
FROM (
  SELECT
    id as letter_id, receiver_id, sent_at, sender_id
  FROM letters
  WHERE
        sender_id=3267 AND sent_at > '2010-03-22'
  ORDER BY sent_at LIMIT 20
) l1 LEFT JOIN LATERAL (
  SELECT
    id as answer_id,
    sent_at as answer_sent_at
  FROM letters
  WHERE
    sender_id = l1.receiver_id
    AND sent_at > l1.sent_at
    AND receiver_id = l1.sender_id
  LIMIT 1
) l2 ON true JOIN human ON (human.id=receiver_id)
```

Let’s use a pagination…

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=1.13..417.82 rows=20 width=47) (actual time=0.049..0.365 rows=20 loops=1)
   ->  Nested Loop Left Join  (cost=0.84..255.57 rows=20 width=28) (actual time=0.040..0.231 rows=20
loops=1)
         ->  Limit  (cost=0.42..82.82 rows=20 width=20) (actual time=0.022..0.038 rows=20 loops=1)
               ->  Index Scan using letters_sender_id_sent_at_receiver_id_idx on letters
(cost=0.42..165.22 rows=40 width=20) (actual time=0.020..0.035 rows=20 loops=1)
                     Index Cond: ((sender_id = 3267) AND (sent_at > '2010-03-22 00:00:00+01'::timestamp
with time zone))
         ->  Limit  (cost=0.42..8.61 rows=1 width=12) (actual time=0.009..0.009 rows=0 loops=20)
               ->  Index Scan using letters_sender_id_sent_at_receiver_id_idx on letters letters_1
(cost=0.42..8.61 rows=1 width=12) (actual time=0.008..0.008 rows=0 loops=20)
                     Index Cond: ((sender_id = letters.receiver_id) AND (sent_at > letters.sent_at) AND
(receiver_id = letters.sender_id))
   ->  Index Scan using humain_pkey on human  (cost=0.29..8.10 rows=1 width=23) (actual time=0.006..0.006
rows=1 loops=20)
         Index Cond: (id = letters.receiver_id)
 Planning time: 0.709 ms
 Execution time: 0.436 ms
(12 rows)
```

Conclusion

Thank you for your attention !

Any questions?

Owly design: zimmoriarty (https://www.instagram.com/zimmoriarty/)

To go further - sources

https://momjian.us/main/writings/pgsql/optimizer.pdf

https://use-the-index-luke.com/sql/plans-dexecution/postgresql/operations

http://tech.novapost.fr/postgresql-application_name-django-settings.html

Slide used with the owls ans paintings from
https://www.instagram.com/zimmoriarty