Leonidas Boutsikaris 2776

Management of complex data
Sunday, 10 May 2020

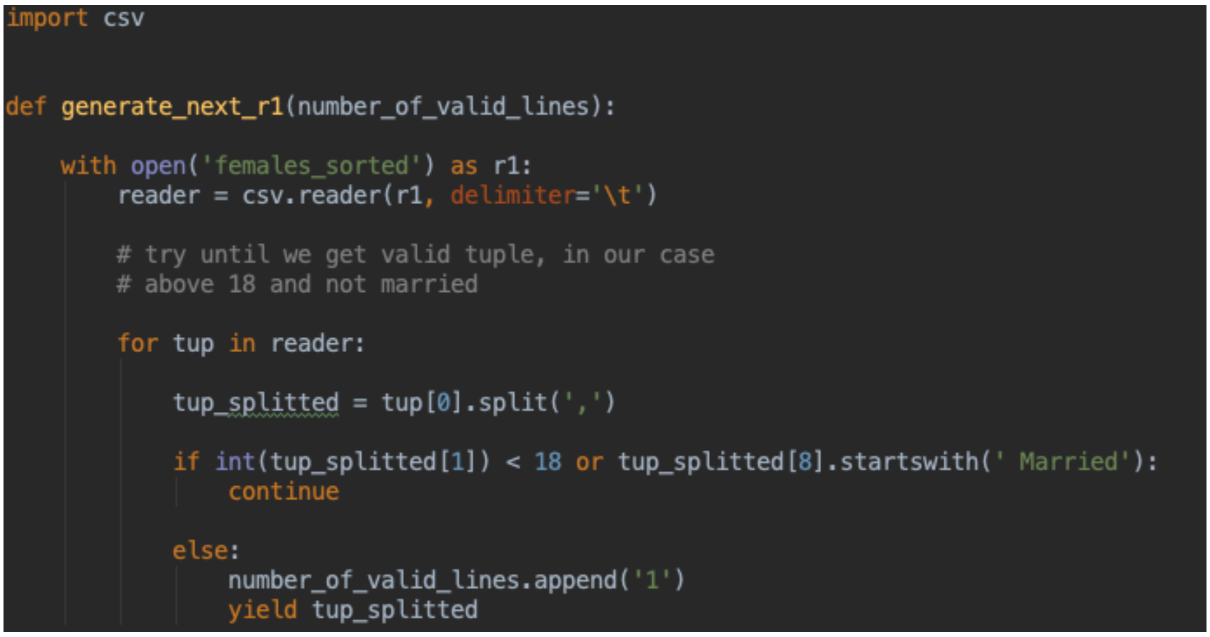
3rd set of exercises

Skyline Top-k queries

We will implement two top-k join algorithms. Data used origin from the income census dataset of the United States in 1994 and 1995. We will use two files, the males_sorted and the females_sorted file. These files are ordered by the instance weight field. We want to find the top-k pairs that have the higher sum of instance weight. These pairs must have the same age be adults and not married.

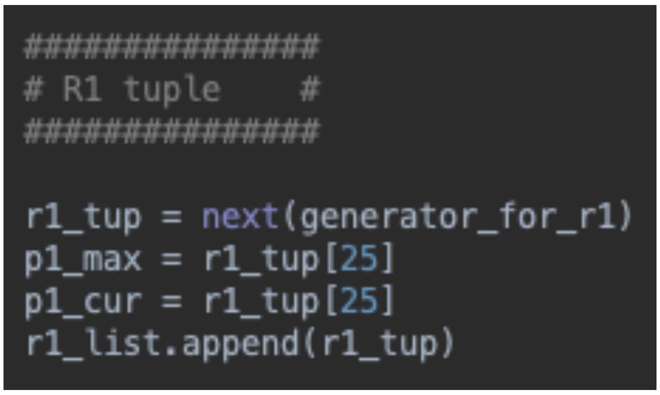
Top-k join A

This algorithm is known as the HRJN algorithm. It uses a threshold to report each result that pass a certain threshold. The threshold ensures us about the outcome. For example, in the first execution, the field of the data we are interested in is in a descending order. Summing the first pair we surely know that we can't have a pair with a higher score. We use two generator functions, one for each relation. The generator function takes as input the data file and returns each time a valid tuple (adult and not married). The same happens for the second relation r2.

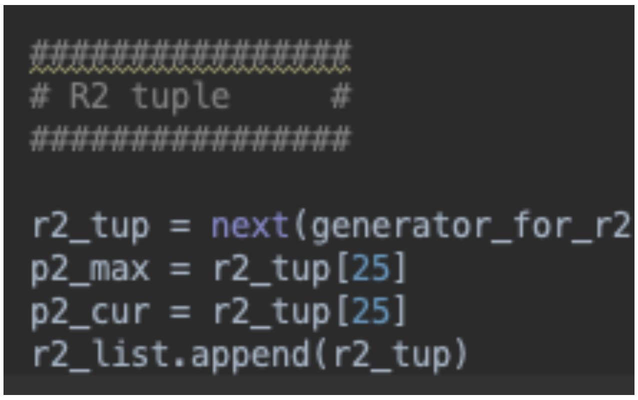


```
import csv


def generate_next_r1(number_of_valid_lines):

    with open('females_sorted') as r1:
        reader = csv.reader(r1, delimiter='\t')

        # try until we get valid tuple, in our case
        # above 18 and not married

        for tup in reader:

            tup_splitted = tup[0].split(',')

            if int(tup_splitted[1]) < 18 or tup_splitted[8].startswith(' Married'):
                continue

            else:
                number_of_valid_lines.append('1')
                yield tup_splitted
```

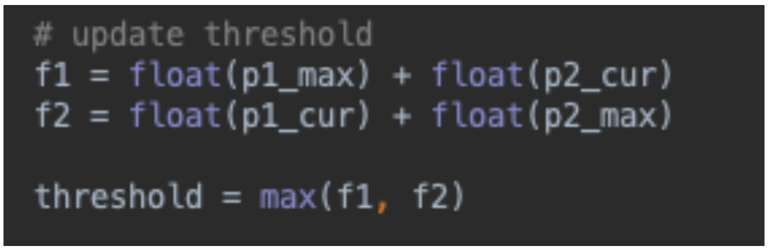
At first our variables are initialized and the threshold is created inside the while loop of the algorithm.



```
################
# R1 tuple     #
################

r1_tup = next(generator_for_r1)
p1_max = r1_tup[25]
p1_cur = r1_tup[25]
r1_list.append(r1_tup)
```

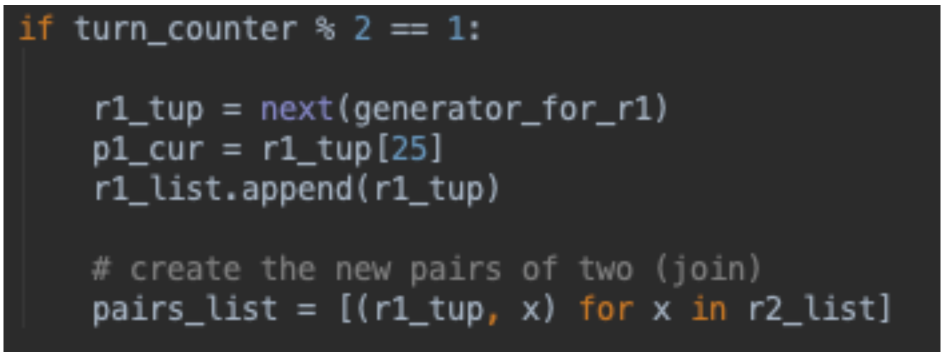


```
#################
# R2 tuple      #
#################

r2_tup = next(generator_for_r2
p2_max = r2_tup[25]
p2_cur = r2_tup[25]
r2_list.append(r2_tup)
```



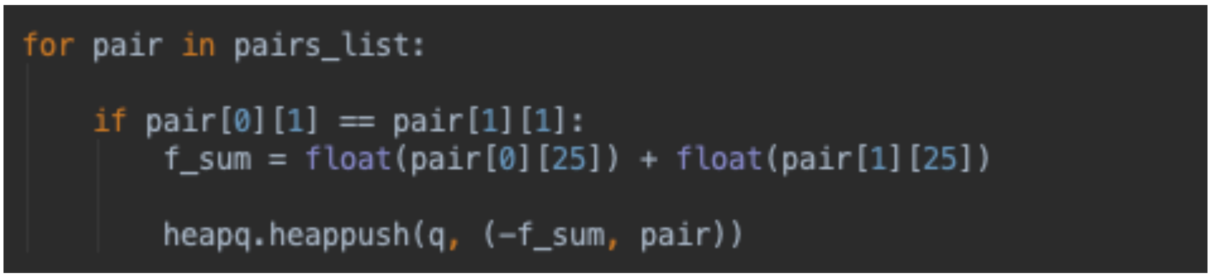
```
# update threshold
f1 = float(p1_max) + float(p2_cur)
f2 = float(p1_cur) + float(p2_max)

threshold = max(f1, f2)
```

Inside the loop, the algorithm takes turns from each file with a mod operator and creates pairs of the current tuple with all the explored tuples of the other file.



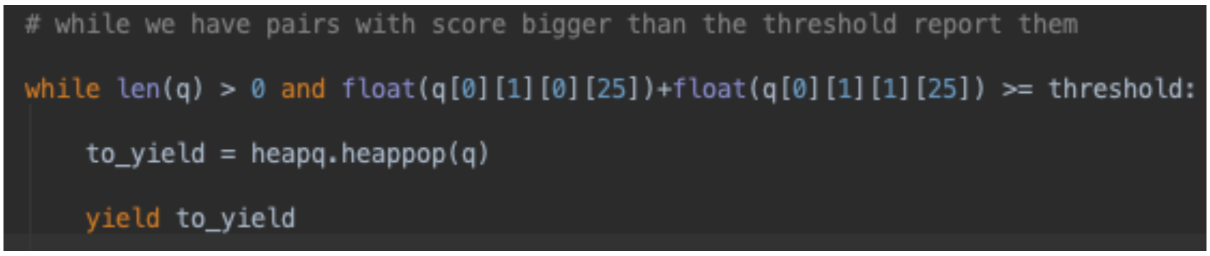
```
if turn_counter % 2 == 1:

    r1_tup = next(generator_for_r1)
    p1_cur = r1_tup[25]
    r1_list.append(r1_tup)

    # create the new pairs of two (join)
    pairs_list = [(r1_tup, x) for x in r2_list]
```

After this step the pairs are checked for age equality. If they pass this condition their score is computed and inserted in the heap. The heap helps us retrieve very quick the biggest or smallest objects we have stored. We use their score as a key and their tuple as their information.



```
for pair in pairs_list:

    if pair[0][1] == pair[1][1]:
        f_sum = float(pair[0][25]) + float(pair[1][25])

        heapq.heappush(q, (-f_sum, pair))
```

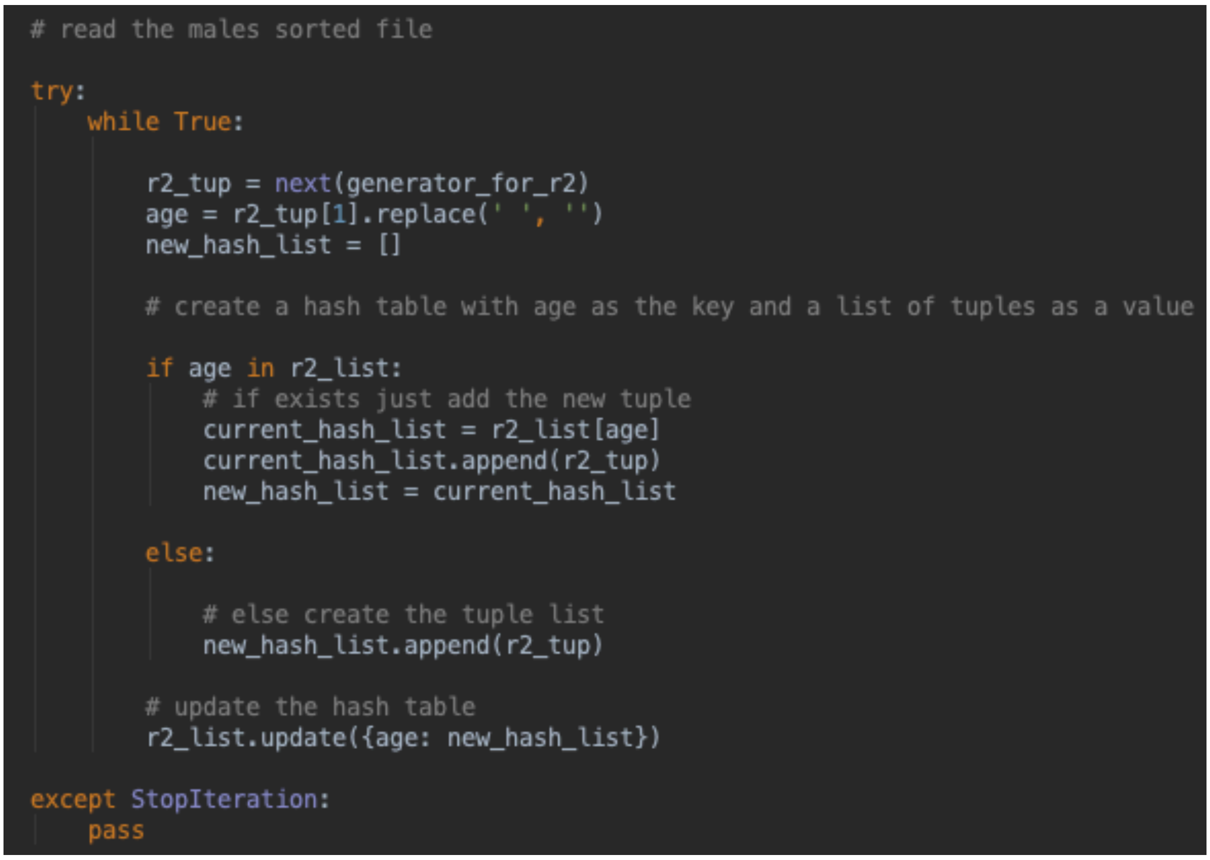
Finally we report every pair that can surpass the threshold and repeat the loop.



```python
# while we have pairs with score bigger than the threshold report them

while len(q) > 0 and float(q[0][1][0][25])+float(q[0][1][1][25]) >= threshold:

    to_yield = heapq.heappop(q)

    yield to_yield
```

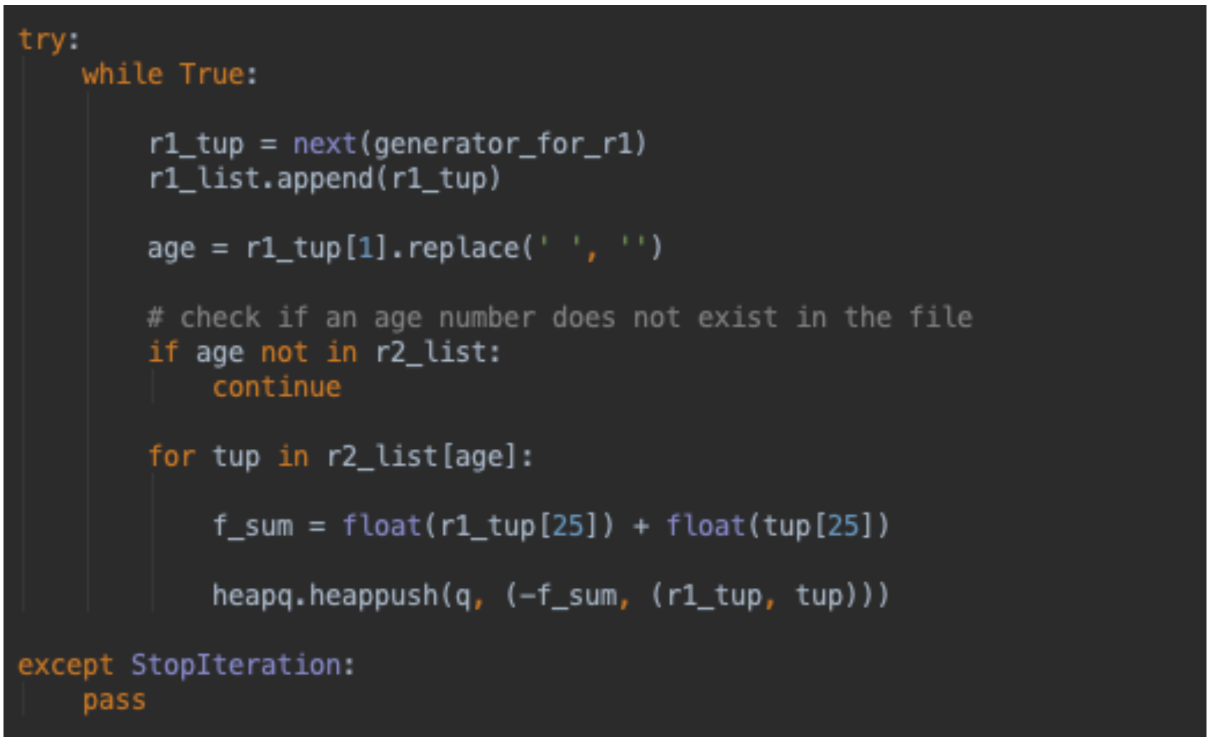
Top-k join B

This second algorithm at first reads and stores in a hash table all the data of the first file.



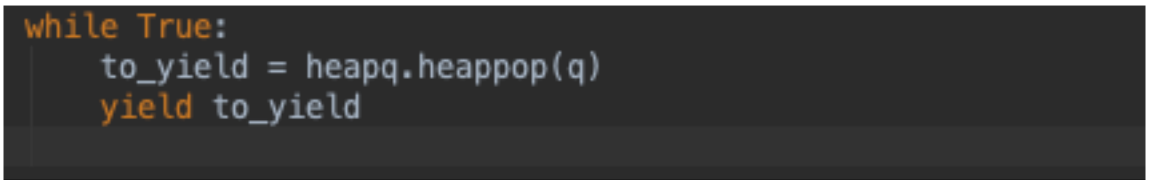
```python
# read the males sorted file

try:
    while True:

        r2_tup = next(generator_for_r2)
        age = r2_tup[1].replace(' ', '')
        new_hash_list = []

        # create a hash table with age as the key and a list of tuples as a value

        if age in r2_list:
            # if exists just add the new tuple
            current_hash_list = r2_list[age]
            current_hash_list.append(r2_tup)
            new_hash_list = current_hash_list

        else:

            # else create the tuple list
            new_hash_list.append(r2_tup)

        # update the hash table
        r2_list.update({age: new_hash_list})

except StopIteration:
    pass
```

After this step it creates pairs of the same age with the other file by using this index.



```
try:
    while True:

        r1_tup = next(generator_for_r1)
        r1_list.append(r1_tup)

        age = r1_tup[1].replace(' ', '')

        # check if an age number does not exist in the file
        if age not in r2_list:
            continue

        for tup in r2_list[age]:

            f_sum = float(r1_tup[25]) + float(tup[25])

            heapq.heappush(q, (-f_sum, (r1_tup, tup)))

except StopIteration:
    pass
```

Finally it reports as a generator all the top-k pairs needed.



```
while True:
    to_yield = heapq.heappop(q)
    yield to_yield
```

Running the scripts:

“***Python3 k_values_graph.py***” - runs automatically for K: 1, 2, 5, 10, 20, 50, 100 and reports back.

“***Python3 main.py k***” where k=number of top-k we want. Runs the two algorithms and reports pairs (both are the same) and running time.

Results

At first we run the two algorithms to check that return the same pairs and have the correct order. The first algorithm is faster in terms of seconds for this specific example (K=5).

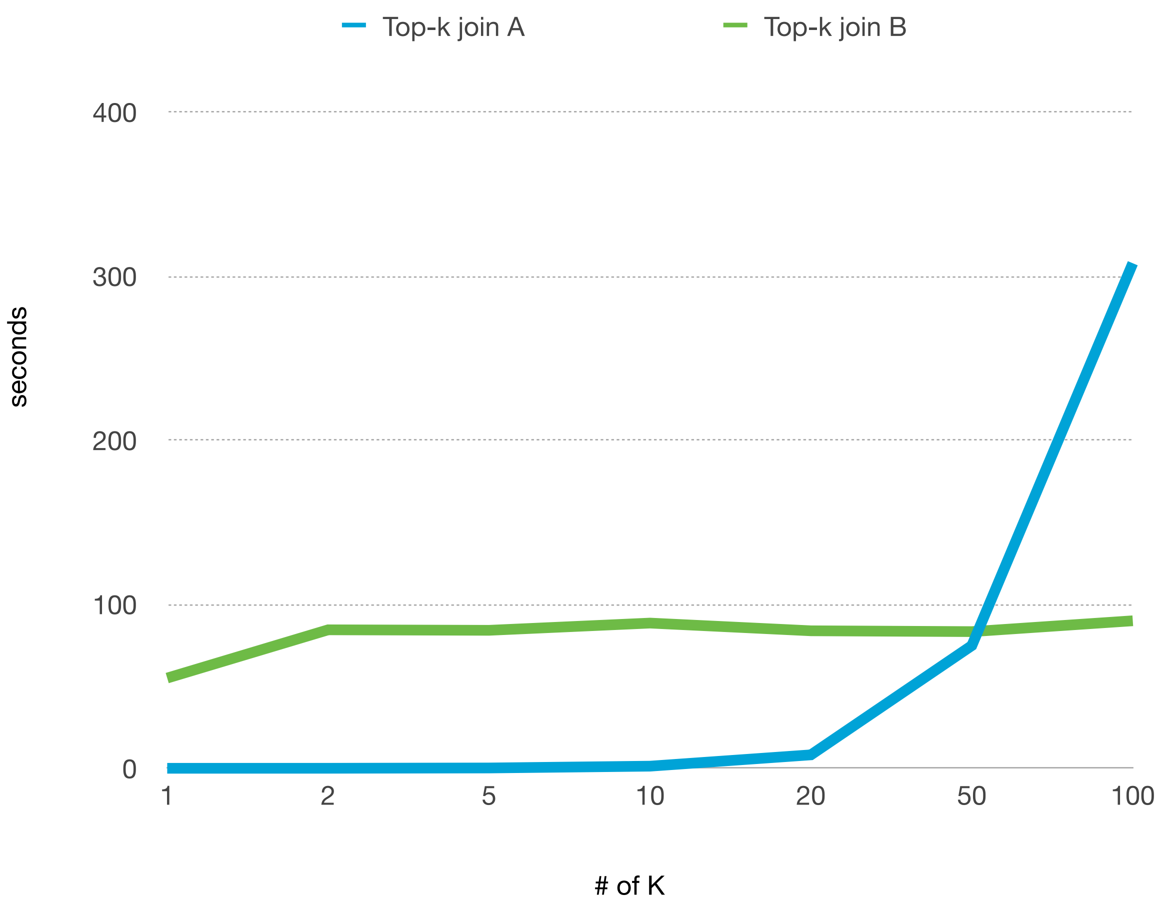
top-k a

```
1. pair: 67141,135085 score:25785.54

2. pair: 44307,135085 score:24247.12

3. pair: 111291,135085 score:23657.66

4. pair: 12112,135085 score:23644.199999999997

5. pair: 183898,135085 score:23046.54

time in seconds: 0.18998312950134277
```

top-k b

```
1. pair: 67141,135085 score:25785.54

2. pair: 44307,135085 score:24247.12

3. pair: 111291,135085 score:23657.66

4. pair: 12112,135085 score:23644.199999999997

5. pair: 183898,135085 score:23046.54

time in seconds: 129.37081694602966
```

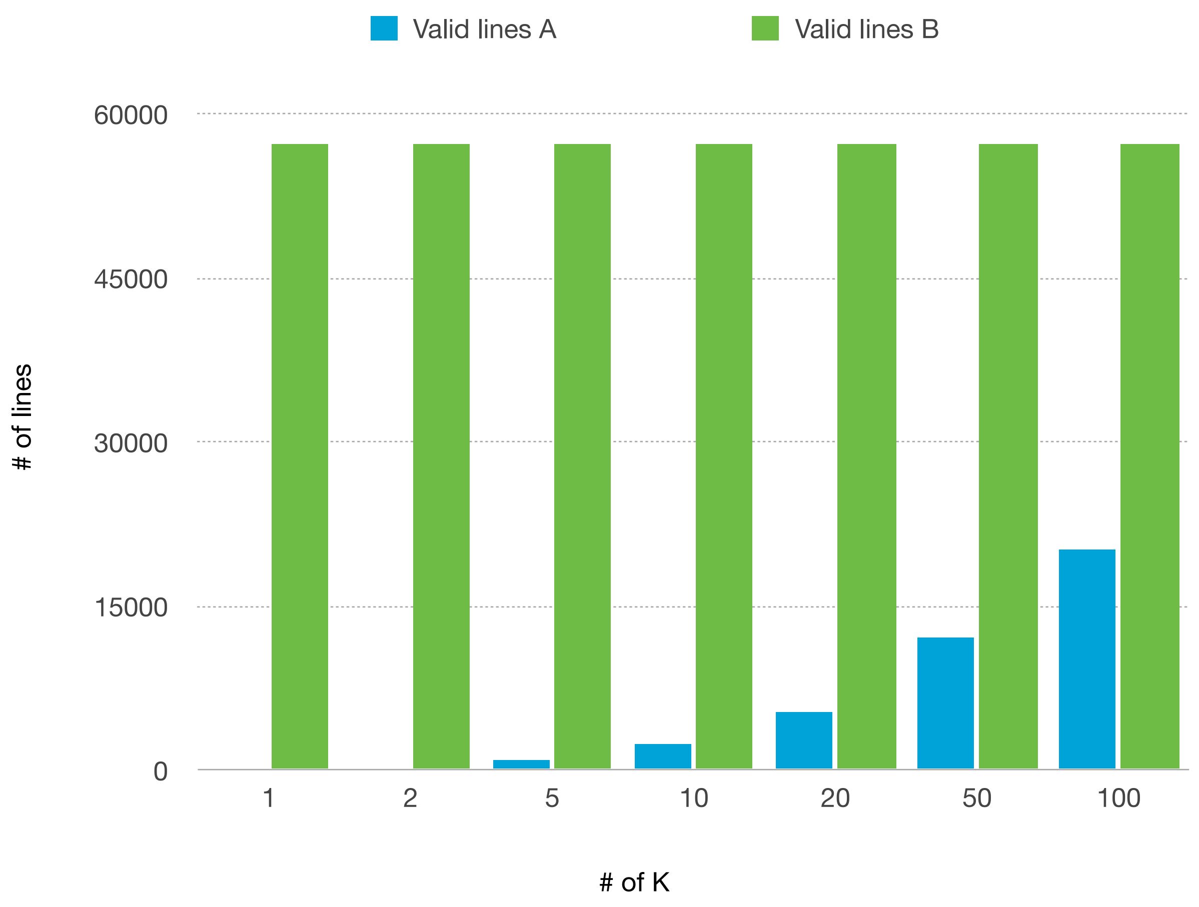
Then we test the algorithms for different values of K: 1, 2, 5, 10, 20, 50, 100.



Top-k join A is really fast when it comes for a small number of top pairs. That is because of the way the algorithm uses the threshold and manages to get the top pairs without exploring all the available data.

The top-k join B has a steady curve. This happens because the algorithm explores all the data and when the algorithm is done the heap has all the top-k pairs ordered. In contrast with top-k join that has to constantly search for the next pair.

We can easily confirm this by looking the valid lines each algorithm read from the files.



To sum up (tldr) skyline queries.

Small $k \rightarrow$ Algorithm A

Big $k \rightarrow$ Algorithm B