CMSC858D --- HW2 --- Caring for your cache with AMQs

==================================

Fatemeh Almodaresi , SID:<117000986>

In [33]:
```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
from decimal import Decimal
import random
import string
import seaborn as sns
```

In [13]:
```
root = "/mnt/scratch1/fatemeh/courses/cmsc858/hw2/"
```

Preparing the input files for both tasks

1. generating the input key files starting from 500k keys to 10m keys jumping 500k with key len varying from 16 to 32

In [12]:
```
minkeylen = 23
maxkeylen = 32
keylen = minkeylen
keys = []
for i in range(10000000):
    keylen += 1
    if keylen > maxkeylen:
        keylen = minkeylen
    r = ''.join([random.choice(string.ascii_letters + string.digits) for
n in range(keylen)])
    keys += [r]
    if i % 1000000 == 0:
        print(i, end="\r")

## Store key files
for i in range(500000, 10000000, 500000):
    np.savetxt("{}/input_files/{}.keys".format(root, i), np.array(keys[:
i]), fmt="%s")
```

```
9000000
```

1. generating query files. all True positives are the smallest key file 500k, all True negatives are 500k punctuations that are not included in making input key files and combining half of each of these, we'll have a query file with 50% positive and 50% negative keys

```
In [22]: keys = []
         for i in range(500000):
             r = ''.join([random.choice(string.punctuation) for n in range(16)])
             keys += [r]
         np.savetxt("{}/queries/negative_500k.queries".format(root), np.array(key
         s), fmt="%s")
```

Holds up for both tasks

1. I've implemented the code in c++17 using cmake project.
2. The code is available online in the public repository of https://github.com/fataltes/cmsc858_hw2 (https://github.com/fataltes/cmsc858_hw2).
3. The number of keys vary from 500k to 1M with jumping size of 500k
4. The fprs vary from 1% to 26% (included) with the jumping size of 5%
5. Three query files are positive (a copy of the first 500k key file), negative 500k keys, and a mix of 250k positive and 250k negative keys.
6. In the two report files submitted in the github repo you can find the construction time and query time for each of the cases for BF and Blocked BF

Task 1

In this task I implemented a bloom filter using the same compact vector library as I used in previous homework. Having the approximate number of keys, n, and desired false positive rate from the user, we can calculate the size of the bloom filter bit vector and the number of hashes. I use murmurhash64 and start it with i*i seed value for i in range(0, num_of_hashes) to generate num_of_hashes different hash values.

I didn't find anything really difficult in implementation of this part except for one thing!!! I need to manually call clean_mem for putting 0 in all bits of the compact bitvector. I originally didn't have that and that was the cause of all the wrong and FPR of 1 as I would go further in my experiments, because compact vector was allocating the same memory over and over where many of the bits were set to one in the previous experiments.

Query Time plots

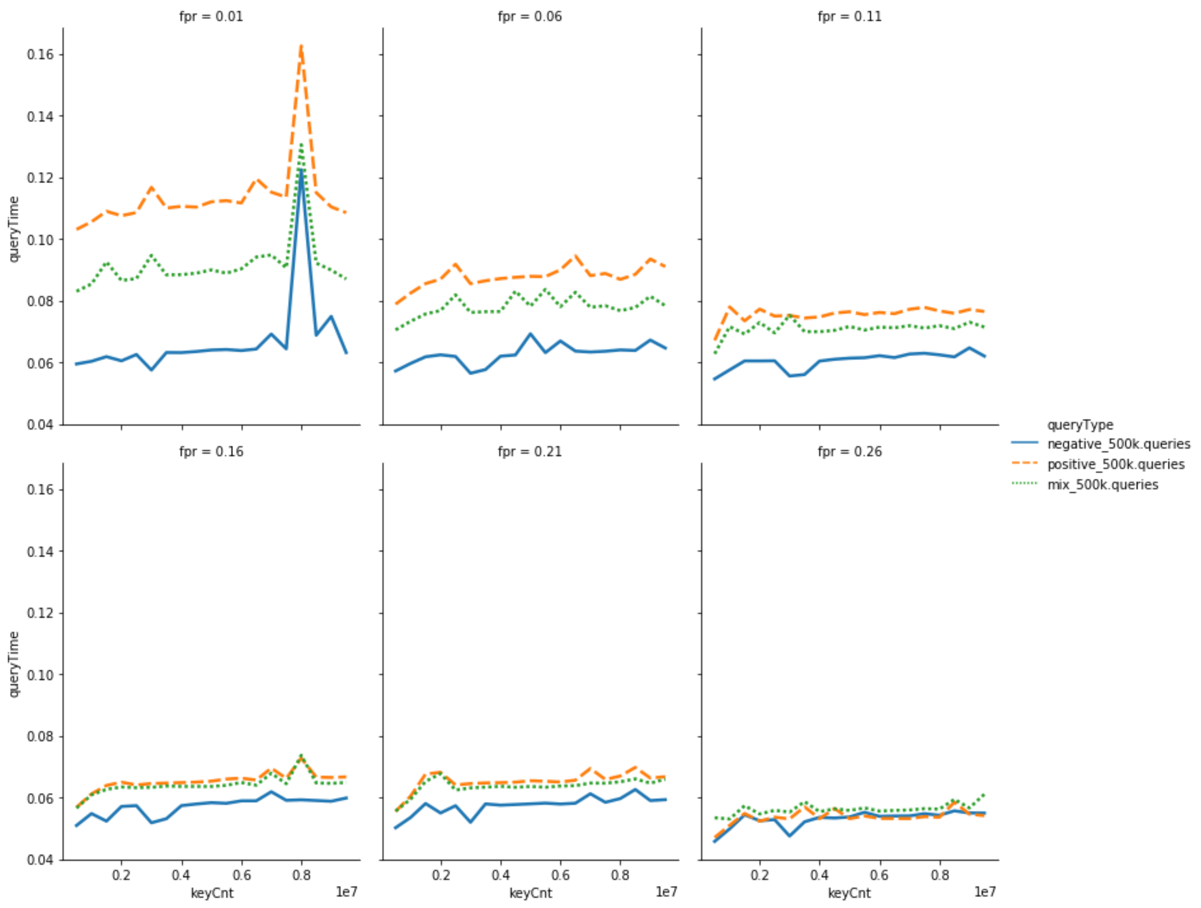
```
In [71]: bf = pd.read_csv("{}/bf_big.report".format(root), sep="\t", header=None)
         bf.columns = ['keyCnt', 'fpr', 'positives', 'queryTime', 'constructionTi
         me', 'queryType']
         bf = bf[bf['keyCnt'] > 10]
```

Below you can see a series of plots showing the total query time over 500k queries (y axis) over bloom filters of different sizes containing different number of keys (x axis) and different FPR (each subplot) for three different types of queries, one with all keys existing in the bf (positive), one with half of the keys available (mix) and one with no keys in the bf (negative) (different hues). (To have the average time per query we just need to divide all the numbers by 500k)

As you can see and as expected, in all the cases, the time is faster for querying negative keys and after than mixed set of keys and it's the slowest for searching the positive keys and the explanation for that is the early stops for any negative key when we get to a bit of value 0. As the FPR increases, the query speeds for querying positive and negative values get closer as we have more FPs that require the same amount of time as TP to query.

p.s. I can't explain the bump in the first plots for fprs 0.01!!! Interesting enough, the peaks and downs are happening at the same key count bloom filter for all three types of queries.

In [72]:
```
sns.relplot(x="keyCnt", y="queryTime", hue="queryType", style="queryTyp
e",
             col="fpr", col_wrap=3,
             height=5, aspect=.75, linewidth=2.5,
             kind="line", data=bf);
```



Experimental FPR vs given FPR

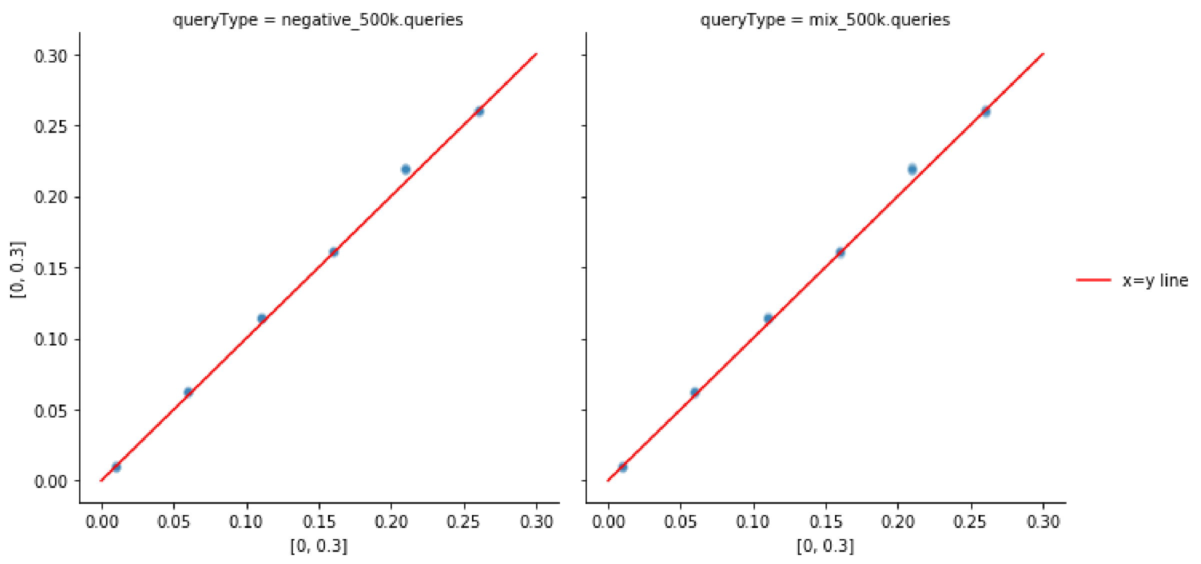
Below, you can find the change of the experimental FPR compared to the desired one.

The experimental FPR fluctuate around the requested FPR. There are multiple points at each FPR, but they are so close and overlapping most of the times.

The behavior is pretty much the same for both querying negative keys and a mixed of both positive and negative.

In [123]:
```
bf['expr_fpr'] = np.where(bf['queryType'] == 'negative_500k.queries', (b
f['positives']) / 500000,
                         np.where(bf['queryType'] == 'mix_500k.queries',
(bf['positives']-250000) / 250000, 0))
p = sns.relplot(x="fpr", y="expr_fpr",
           col="queryType", col_wrap=2,
            height=5, aspect=.95, alpha=0.1,
            data=bf[(bbf['queryType'] != 'positive_500k.queries')]);
plt.plot()
p.map_dataframe(plt.plot, [0, 0.3], [0, 0.3], '-', label='x=y line', col
or="r").add_legend()
```

Out[123]:
```
<seaborn.axisgrid.FacetGrid at 0x7f5c0a5317d0>
```



Task 2

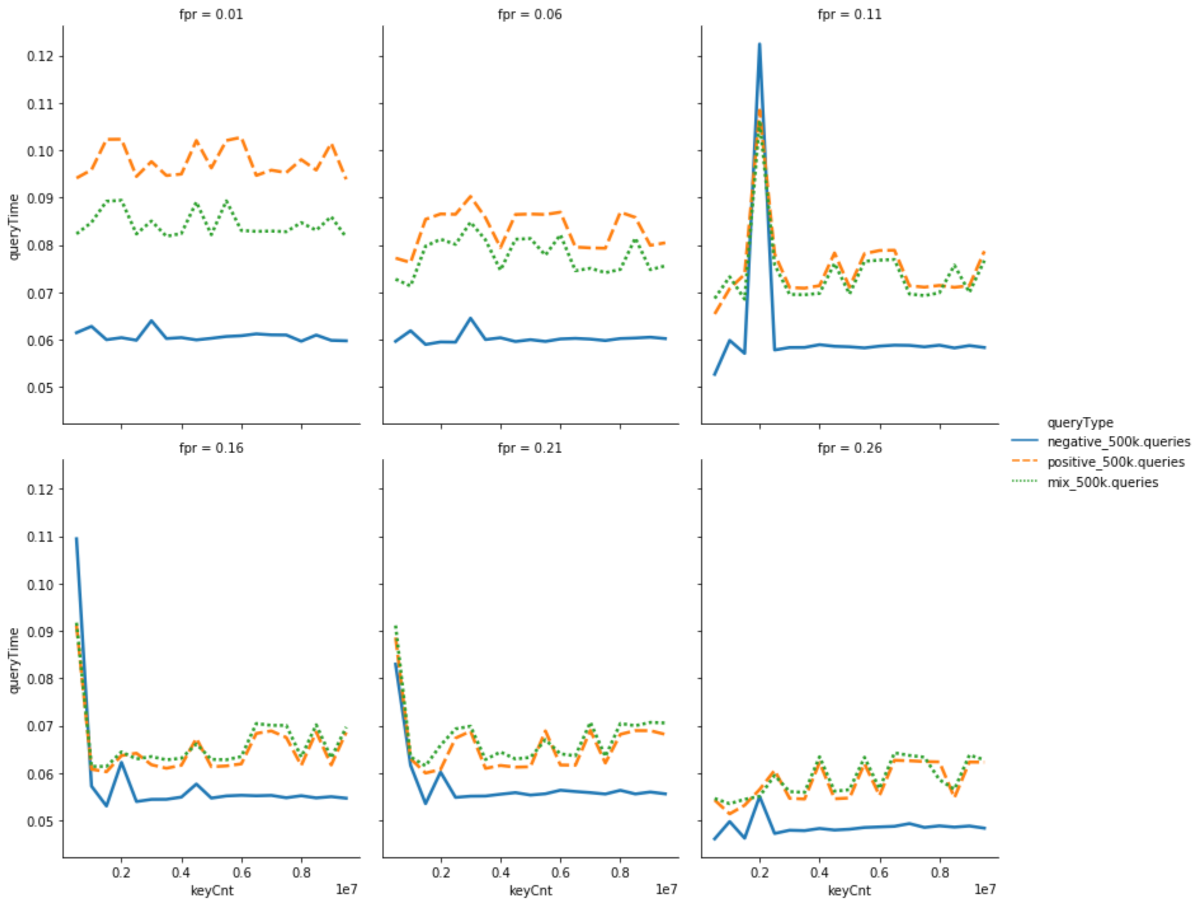
In this task I implemented a blocked bloom filter using the same idea as bloom filter in the previous task except that as explained by the paper, I use the first hash value to choose the block and the rest to walk in that block. I've fixed the block size to 512 bits. The whole data structure is still a single bitvector which I consider as blocks and use the first hash function to jump to a specific cache-fit block of that bitvector. I should note that we do not need to have separate data structures to fit the cache because as long as we are not passing the limits of a cache size while reading a chunk of memory that memory stays in cache.

I think the main difficulty for me in this section was actually understanding the equations in the paper suggested for description of blocked bloom filters. I still doubt that I have understood and implemented the idea correctly and I'm not sure I exactly understand why the false positive rate might be a little bit worse than the basic bloom filter and what was exactly the parameter "*c*" in the paper that we could change to tradeoff between time and fpr!

Both query time plots and fpr plots show the same behavior as for Bloom Filter except that we see only one bump in FPR=0.01 and nothing in other plots. That makes me wonder if that is the point where the Bloom Filter passes the cache size or some sort of system memory cache related explanation for these peaks that don't show up in the block bloom filter implementation anymore.

Query Time Plot

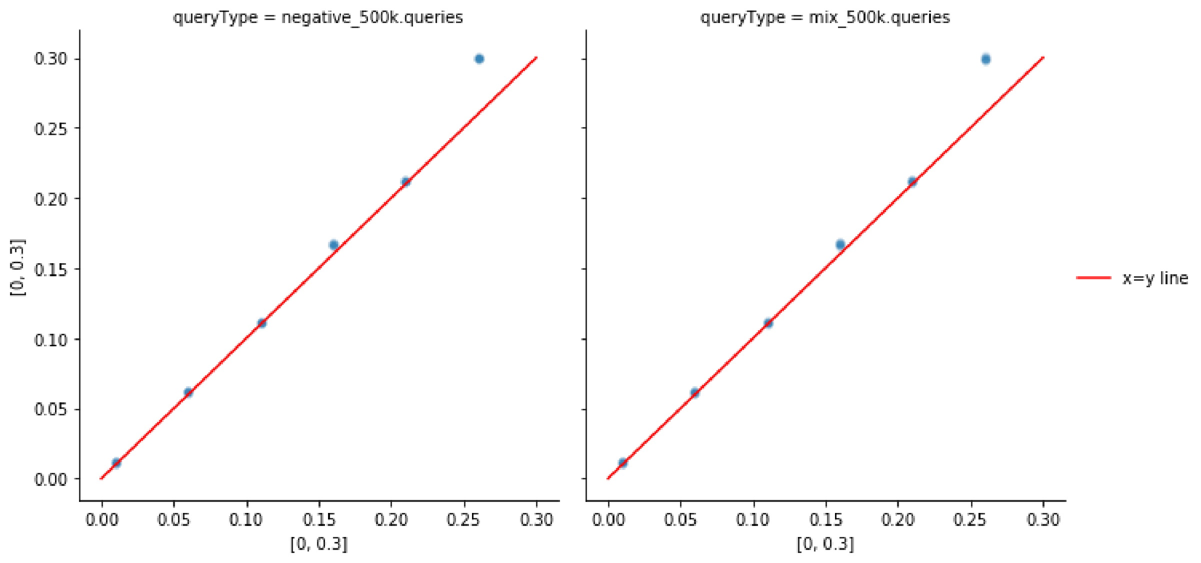
In [121]:
```
bbf = pd.read_csv("{}/bbf_big.report".format(root), sep="\t", header=None)
bbf.columns = ['keyCnt', 'fpr', 'positives', 'queryTime', 'constructionTime', 'queryType']
bbf = bbf[bbf['keyCnt'] > 10]
sns.relplot(x="keyCnt", y="queryTime", hue="queryType", style="queryType",
            col="fpr", col_wrap=3,
            height=5, aspect=.75, linewidth=2.5,
            kind="line", data=bbf);
```



At the FPR of 0.26, the experimental FPR goes higher than normal and gets close to 0.3 which is a proof on BBFs having a slightly worse FPR compared to BF.

In [122]:
```
bbf['expr_fpr'] = np.where(bbf['queryType'] == 'negative_500k.queries', 
(bbf['positives']) / 500000,
                         np.where(bbf['queryType'] == 'mix_500k.queries'
, (bbf['positives']-250000) / 250000, 0))
p = sns.relplot(x="fpr", y="expr_fpr",
           col="queryType", col_wrap=2,
            height=5, aspect=.95, alpha=0.1,
            data=bbf[(bbf['queryType'] != 'positive_500k.queries')]);
plt.plot()
p.map_dataframe(plt.plot, [0, 0.3], [0, 0.3], '-', label='x=y line', col
or="r").add_legend()
```

Out[122]:
```
<seaborn.axisgrid.FacetGrid at 0x7f5c0a55a950>
```



Comparing Bloom Filter and Block Bloom Filter

Time and FPR

In this last section, I concatenate the results of the two data structures of BF and BBF and by factoring out some parameters we had in our experiments, try to see how their results compare with each other in query time and FPR.

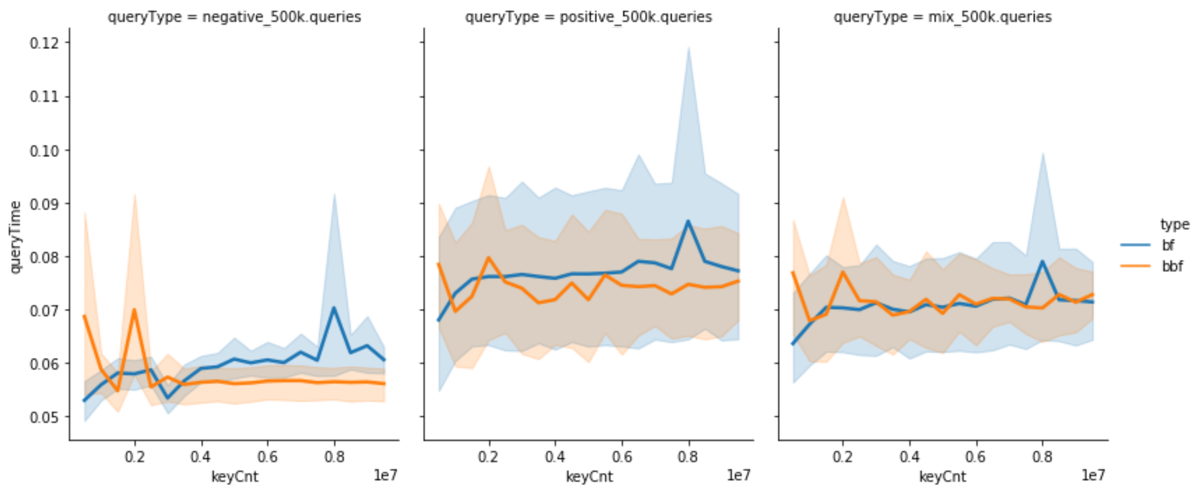
In [125]:
```
bf['type'] = 'bf'
bbf['type'] = 'bbf'
joined = pd.concat([bf, bbf])
```

Query Time comparison of bf and bbf

As expected, we can see that the query time for bf is usually higher than query time for the same set of queries over bbf although, different FPRs of both cases make some overlaps of the intervals of the two distributions. This is however explainable as like what we observed earlier, there is another trend for the increase in query time wrt the FPR increase.

Also, here we can see the trend of increase in query time as we have more and more positive cases more clear.

In [126]:
```
sns.relplot(x="keyCnt", y="queryTime", hue="type",
            col="queryType", col_wrap=3,
            height=5, aspect=.75, linewidth=2.5,
            kind="line", data=joined);
```

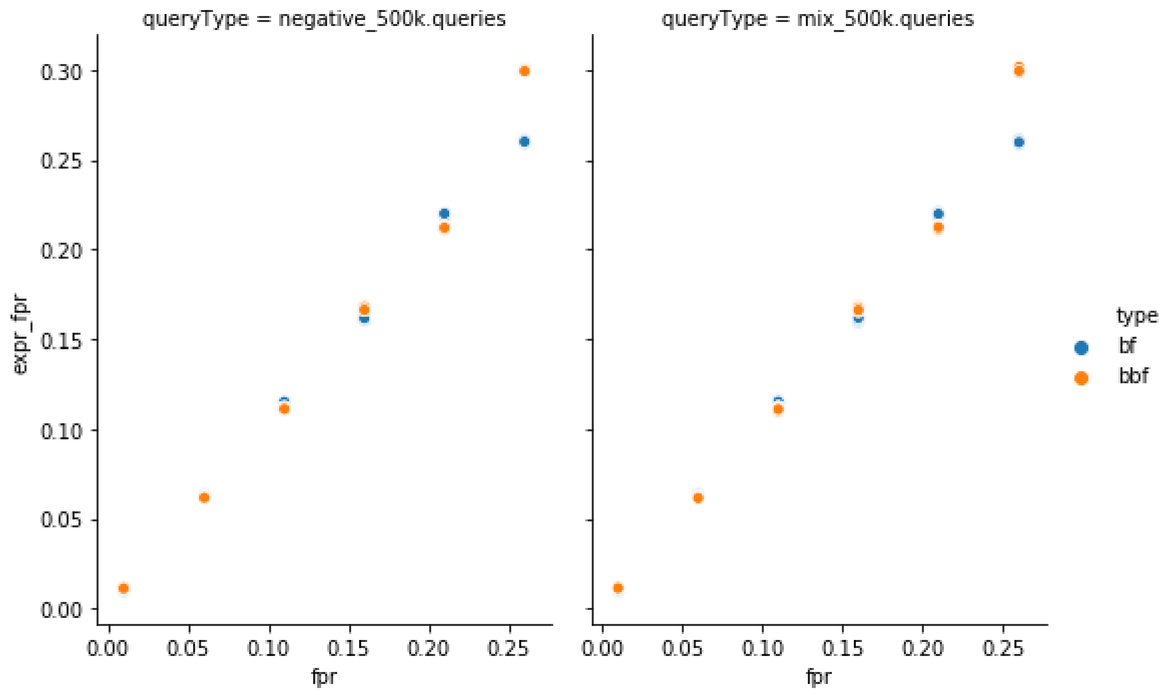


FPR comparison of bf and bbf

Here, we can see that the FPR for BBF is only slightly worse for smaller FPRs and even sometimes better for larger ones. This is a beautiful observation and aligned with what was explained in the paper.

In summary, using BBF, we can have better query time benchmarks with almost no harm to the FPR and size of the data structure.

In [132]:
```
sns.relplot(x="fpr", y="expr_fpr",
            col="queryType", col_wrap=2,
             height=5, aspect=.75,
             hue='type',
             data=joined[(joined['queryType'] != 'positive_500k.queries'
)]);
```


queryType = negative_500k.queries
queryType = mix_500k.queries
expr_fpr
fpr
type
bf
bbf