# Big points

* **Exploit space efficiency and constant look-up times of bloom filters** for approximate/exact searching.
  + Insert probable variations of words (e.g., likely 1-edit errors)
  + Insert all n-grams of a document up to size n (exact matches on n-gram terms)
  + Use other pre-processing techniques, e.g.: (facilitates approximate matching + dimension reduction)
    - Lexographic ordering on n-grams
    - Locality-sensitive hashing, e.g., soundex
* **False positives.** The more items in a bloom filter, the greater the probability of a false positive (the limit being so many items are inserted into the bloom filter that every hashable index location is true – at which point it says everything is a member). How to address this? Machine learning + big data.
  + Learn bloom filters that are more willing to generate false positives on unlikely n-grams, and less willing to generate false positives on likely n-grams.
    - Exploit inherent regularity of language
  + Simple tricks, e.g., a bloom filter may only have n-grams up to size k, so anything above that is immediately rejected.
* **Metrics**
  + Many different metrics, and none uncnoditionally the best. That said, we’re primarily interested in effective measures that have already been demonstrated by others. By effective, we mean gives users the ability to find the particular data they’re looking for without too much searching.
  + Keyword/term weighting. Some terms are more discriminating than others. For example, a term that is in every block does not tell us much. A term that is in only one of the blocks tells us a lot. The latter is more discriminating—we know we only need to look at one of the blocks to find the keyword as opposed to having to search through all the blocks to find it. So, weight keywords/terms by their ability to discriminate, e.g., frequency in blocks, the higher the frequency the lower the weight.
    - One weighting scheme is –log(n/N), where n is number of blocks with the term and N is total number of blocks. Works well in practice.
  + Proximity/locality metrics. What sort of proximity measures make sense? The idea is that, yes, a block may have a number of keywords, but what if it is really far from other keywords in the search? Does that matter?
    - There are many different measures to use here. Here’s an easy but effective one to rank blocks (if N blocks, we will rank them from 1 to N):  
      Block cost = , where weight is the previously described keyword weighting approach and distance is something like “how many blocks away is the nearest occurrence of the keyword?” The higher the cost, the worse the block is. We can normalize it to be between 0 and 1 and color it accordingly. This scoring method matches up well with the visualization component of our research.
  + Outlined other approaches also. The above two metrics may be more useful, but I can still explore more approaches.  
    *The most complex approach: treat the problem as a logical graph search problem to find a good state, where good is defined as how closely the best state matches the exact query–the closer it is to matching it exactly, the closer it gets to a max score of 1. The closeness (similarity) score can factor in things like permutation of words (how closely does it match query order), how many gaps between the keywords in the document there are (approximately how many gaps in the case of blocks and precisely how many in the case of n-grams & n-grams with gaps), and so on. It could find a best match for a query like “the quick brown fox jumped over the lazy dog”, e.g., maybe in block 2 the words “jumped” and “fox” appear, and in block 4 the 2-gram “quick brown” appears, and in block 5 the 3-gram “over lazy dog” appears. Apply distance metrics to this to get a measurement about how close this is to matching the exact query. I’m not sure this is a reasonable metric, though. It also has a* ***huge*** *state space to explore, but local search methods can be used to find quick solutions that achieve local minima.*

# Details

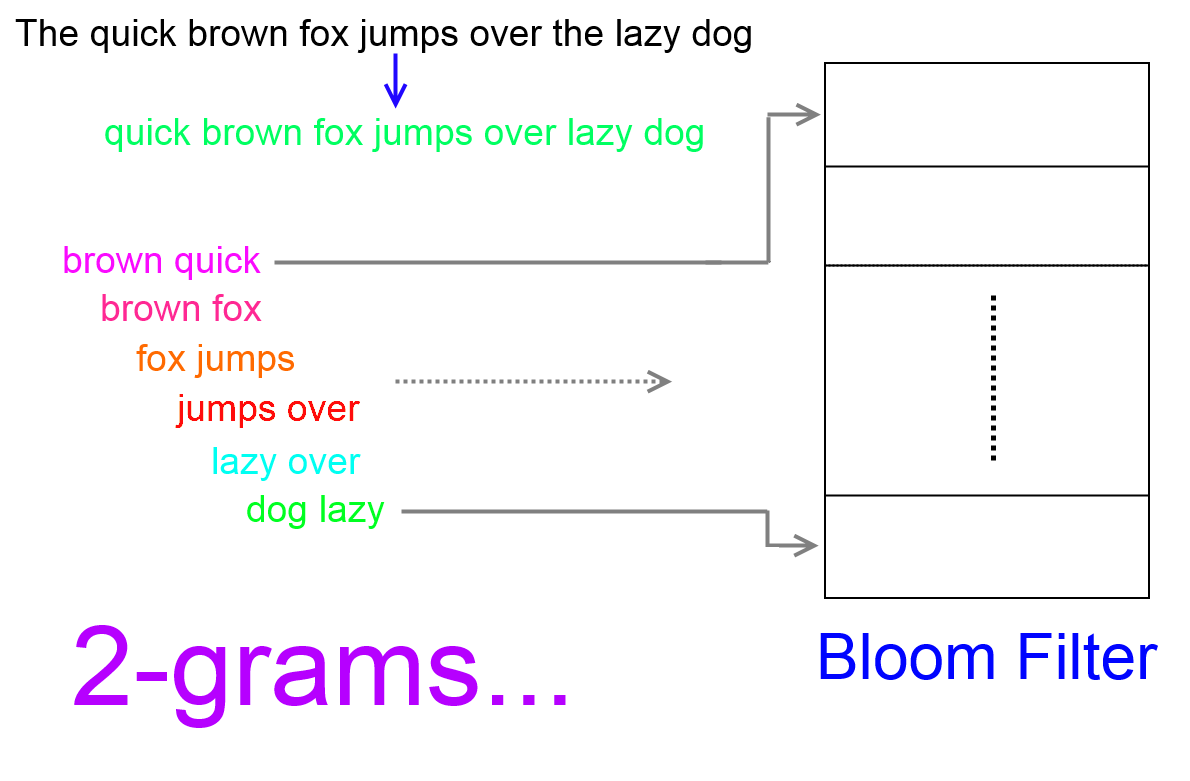
First some terminology. A search query is composed of a set of terms (keywords).  
  
For example, searching for ***hello beautiful world*** has three terms, “hello”, “beautiful”, and “world”. When searching, we will try to find occurrences of each one of those terms independently.  
  
A term consists of either a single world (a 1-gram), or multiple words enclosed in quotes, e.g., “hello beautiful world” (a 3-gram). This 3-gram counts as one term, and when searching for this term we will look for that exact (or approximate, depending on search type) phrase. An example of this? Google searches that have quotes around them.  
  
All of this can be combined into a single search query, e.g., searching for: “beautiful planet” “hello earth” alex towell. This search query consists four search terms, each term given a different color. In searching, we will try to find occurrences of each one of those terms independently.

**Bloom filters**’ are space efficient, probabilistic data structures for representing sets—they answer the question, is something in a set? The down-side is, they have a non-zero probability of false positives. For a given bloom filter, the more you add to it, the more false positives you get. (The limit to this? Everything tests positive.)

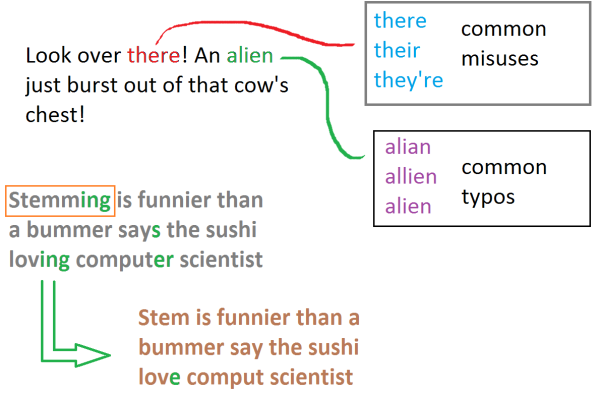
To take advantage of the space efficiency of bloom filters and to facilitate rapid and approximate searching, let’s put the bulk up the work upfront--when constructing the bloom filter. What’s this mean?

* Insert 1-grams, 2-grams, …, n-grams for exact mathces on search terms up to n-gram terms.  
    
  For example, the following keyword search has two terms:

**“fox jumps” quick**

To see if there is match, this just requires two constant (O(1)) looks up, one for “fox jumps” and one for “quick”.  
  
For approximate n-gram matches, sort n-grams lexographically before inserting them into the filter. And, on user queries, do the same thing. Order is thus ignored.  
  


* Insert common variations, e.g., typos/misspellings (perhaps all 1-edit errors!) to make approximate searching more forgiving of malformed input without sacrificing speed.
* Other pre-processing techniques, e.g., stemming (where we reduce a word to its root) or soundex (words that sound alike hash to the same value—a form of locality-sensitive hashing).   
    
  These pre-processing techniques in particular have the added advantage of reducing the number of unique members (dimension reduction)—which means a simpler bloom filter.



Ok, but what about the ***false positives***? The more you put into it, the greater the probability of a false positive, right? Yes, this is unfortunately true. However, there are a couple things we can try to do to mitigate this:

1. Reduce the language-space – fewer members, smaller chance of a false probability for a given bloom filter size. Some of the techniques we’ve already discussed (e.g., locality-sensitive hashing). Other techniques include:  
   1. Language-space is infinite, but the space represented by the bloom filter doesn’t need to be. For instance, record the n-gram size (n = 1, 2, …, n) a bloom filter includes as members, and don’t even consider anything that has a query term larger than n-grams.  
        
      For example, a bloom filter may only include 1-grams and 2-grams. Therefore, we immediately know any term that consists of more than two words is not in the filter. We may give the user the option to reduce the term into smaller chunks, but that’s a different topic.
   2. …
2. To minimize false positives, we are better off minimizing false positives on probable search queries—that is, probable n-grams. Language is very structured and infused with regularity. Having a false positive on “***jw82bzlsggddd0816644kv***” is probably not a concern, but having a false positive on “***hello world***” is.  
     
   Mathematically:   
     
   Where **P[fp]** means “probability of a false positive”, **P[fp | ngram]** means probability of a false positive given a certain **ngram**, and **P[ngram]** is the probability of the ngram. If **P[ngram]** is a very small number, then it won’t affect P[**fp**] much.   
     
   So, how to do this? Big data to the rescue!  
   1. Make an n-gram distribution from a relevant text corpus, where each n-gram is weighted by its frequency.  
        
      What’s a “relevant source of text”. This can be anything, from a large collection of books to an enterprise’s collection of emails (privacy?). Before generating the n-gram distribution, we may want to preprocess the corpus in a certain way, e.g., removing stop words, stemming, etc. And, of course, ideally the corpus will include typos and other typical variations of words. [Note: on the one hand, we could handle typos at the interface level, e.g., automatically fixing spelling errors; on the other hand, an interesting research question is to see how much of this work can simply be delegated to the bloom filter so it can do it in constant time.]  
        
      How we choose/transform a text corpus will probably play an important role in performance, e.g., minimizing probability of false positives or learning a ***simpler*** bloom filter. In theory, ***complexity*** is measured with respect to the minimum size of the description [for our bloom filter—including the data and the hash functions], but we’ll use simpler measures of complexity. More on that later.
   2. Sample (non-uniformly, according to frequency) examples from it. If the n-gram example is actually in the text (positive example), no harm no foul. If it’s not (it is a negative example), then if the bloom filter reports a false positive on it, count that as a failure.  
        
      (Alternatively, we can consider every n-gram in said distribution, but penalize a false probability proportional to the ngrams frequency. However, if the ngram distribution is large, then this might not be feasible.)  
        
      In learning the parameters for the bloom filter, our objective is to minimize the **number of failures** on the given n-gram distribution.
3. **Over-fitting**. The n-grams we sample during the training phase (described in the previous point) is called the training set. During training, we are trying to minimize the probability of a false positive on the training set, but there is a risk of too much over-fitting to the particulars of the training set [that is, fitting to the particular patterns in the training set which do not in general extend to the entire population].  
     
   That is to say, our bloom filter may do splendid on its training set after having learned parameters that provide it with the least probability of false positives, but it may do poorly on a different but equally legitimate n-gram training set.  
     
   To mitigate this, we have some options:
   1. **Occam’s razor**. Include a “specialization” penalty, e.g., having n hash functions is, all other things being equal, worse than having (n-1) hash functions. So, the total cost of the learned parameters do not just include the probability of a false positive, but also include the cost of the complexity of the bloom filter. This gets back to the brief earlier discussion on complexity.  
        
      What are some parameters we can penalize? The square of the number of hash functions, the square root of the size of the bloom filter, etc.
   2. We can see how our training bloom filter performs on a validation set, in which we again sample from an n-gram distribution, but maybe a **different** distribution or just the same distribution but due to randomization we will select different samples in different proportions. This answers the question, how well does it generalize to unseen examples?
   3. We can combine the two previous points into an algorithm that takes both into account. Here’s an outline.
      1. We train our bloom filter in the order of least complex to most complex, e.g., least number of hash functions to largest number of hash functions.
      2. Train the bloom filter, with the given complexity level, on a training set and keep track of the top N performers.
      3. When finished training, see how well the top N performers do on a validation set. Find the one that does the best.
      4. If this is not the first iteration, see how well the best one during this iteration did compared to how well the best one had done during the previous iteration. Do we see an improvement? If so, then we are not overfitting yet. Increase the complexity level and go back to step 2.
      5. So, it’s starting to get worse. This is evidence of overfitting. We can stop training now; the bloom filter learned during the previous iteration is the best. So, we’re done. Return it as the optimal bloom filter.
   4. Letter n-grams. Use this to make up words, where the probability of seeing a letter is a conditional probability: P[letter | previous n-1 letters]. We can use this language model to generate somewhat regular looking words or very irregular looking words, depending on the size of n and depending on the text corpus of data we derived letter frequencies from. We can use this to generate a new text corpus that follows whatever model we used, and then we can, like in previous steps, sample word n-grams from that distribution.  
        
      Why? So that we do not over-train on “well-formed” queries. We still want to work on somewhat malformed queries. Of course, if the text corpus we use to generate the word n-grams from in previous steps already has such malformed data, then it may be best to rely upon its more realistic distribution.  
        
      The worst case is when the letters are uniformly distributed. In that case, there is no regularity whatsoever. If we train our bloom filter on that, we are back to the worst-case scenario where we try to minimize the probability of a false positive without considering the probability of seeing a particular n-gram. This will assuredly do worse, but it would productive to see how much worse.
4. **Genetic algorithms.** A note on training implementation. One (somewhat bad, probably) option is to make bloom filters with random parameters (but at a specified complexity level, e.g., a certain sized bloom filter and/or a bloom filter with a certain number of hash functions). This may work well enough, actually. However, it is far from an ideal solution since it converges very slowly to good solutions. To remedy this, a simple adaption of the previous algorithm is to save the last N top performers, and when trying a new bloom filter, with some probability choose one of the best N and change it in some way. To avoid getting stuck in local minima, you won’t necessarily always choose from the top N, sometimes you’ll just generate a completely random one.  
     
   Taking this approach to its limit, we have what are called genetic algorithms. Find some way to encode the bloom filter parameters such that two different encodings (parents) can be combined in some way to generate a new bloom filter (child) that has properties derived from both parents. Occasionally, mutate a child (say, change a coefficient or an operator in one of the hash functions). Now, introduce a bunch of these into the population, and choose to mate pairs in accordance with how “fit” they are (the lower the probability of a false positive, the more fit) to generate a new population. Rinse and repeat until there are signs of over-fitting, as previously discussed. At that point, we have converged to a good solution.  
     
   I already have the random (uninformed) approach working. I’ll be trying to improve upon this, however, by using a genetic algorithm. Maybe it makes a difference, maybe it doesn’t.

# Discussion on previous points you made

(1) Depth in the technical details: For some issues discussed in your manuscript, what we can do to include “depth” would be to analyze their impacts. For example, for n- search, we can measure the impact of increasing “n” in “n-grams”. By saying “impact”, the two factors I would suggest is (a) search speed and (b) data size (in bytes). However, just measuring the two metrics really do not give us how good (or bad) the performance is. To take care of this issue, it may make the most sense to compare the performance of our Bloom-filter based solution to the index-based solution proposed by Navarro (in “Adding Compression to Block Addressing Inverted Indexes”).  
  
The size of the n-gram will have an effect on the speed of query answering for some of the approximate search methods I explored. The one I am primarily interested in now, however, is not effected. However, it will increase the size of the bloom filter.

For instance, we could create a bloom filter for the entire text with n-grams from size 1 to size n. This means that a bloom filter will have a factor of n more members (the number of characters, or words, in an individual member is immaterial). Each member, for a given false positive probability (fp), on average needs 1.44 log 1/fp of bits per member. We'll try to do better than this -- see the discussion on training.

(2) Existing work: a chapter of describing existing work is another component we can add. By describing what existing solutions have been proposed, we can describe/convince the audience the contribution of our work. I have reviewed more than 20 papers and I can help you for this. The byproduct of this effort (describing the major existing solutions and the uniqueness(es) of our solution) is a list of references, which is a required component in a thesis.

Adding the above two contents to your draft, your draft can be ready as a thesis. Regarding (1) above, it is not necessary for you to study the impact to the performance about every issue you discussed in your draft.

We should discuss this more, especially in light of Dr. Yu’s request for a background survey on existing research in the area.

# Another idea (separate and unrelated to the preceding details)

Blocks do not have to exist as parts of a document. Like in that paper, “Adding Compression to Block Addressing Inverted Indexes”, blocks can cover any desired granularity, and we can have multiple heirarchies of blocks. For example, consider a binary tree of blocks. To localize, at the smallest block granularity, the keywords in a document space, there are only log2N levels to check, N being the number of blocks. Since the left and right blocks for a given parent may both match or come close to maching approximately, we could explore both children if we didn’t want to discriminate between them yet. This is an “embarssingly parallel” problem since searching through each sub-tree is an independent problem that can be done concurrently. At the top level of the tree, we have an all encompassing block of the entire document space. Quickly, in constant time, we can deterimine if the keywords are anywhere to be found. Next, we’ll check out the left and right children to see which one of those matches. Rince and repeat until we have got to the block granularity we desire. Each step can be constant O(1) until we get near the bottom of the tree at which point we can apply more distance metrics to be more discriminating between blocks.  
  
Note that we don’t even have to reveal the structure or content of the document, so this document space can be encrypted, and only expose a search interface to the bloom filters, and the bloom filters can be tuned to give as much, or as little, context/detail as desired.