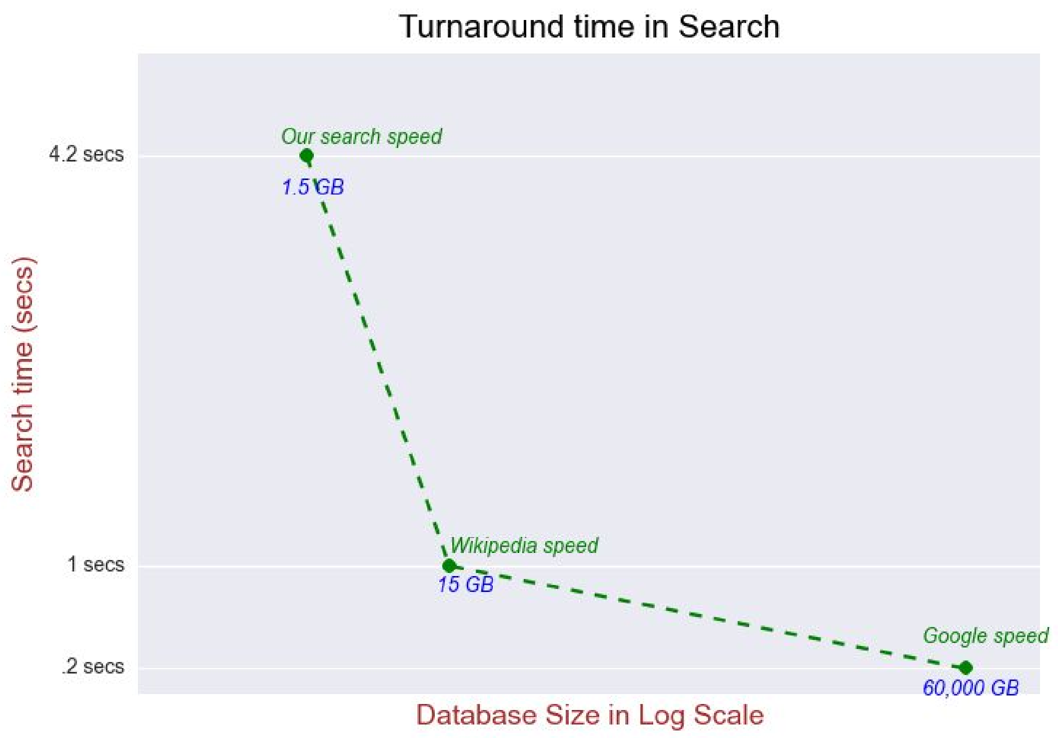
Search Indexing in an Elastic Database

Swetha Reddy, Tate Campbell, Abhishek Singh

UNIVERSITY OF SAN FRANCISCO

Master of Science in Analytics

Why Search Indexing



1. There is now massive amount of data not just on the web but also in many production level databases

2. Efficient Information retrieval with highest throughput & lowest cost has always been the top priority when using such databases

3. Search Indexing comes in as a solution for any such application

Database Creation

1. Created a 1.5 GB Text corpus (4 Mn documents) using python
2. **API based** data pull from NYTimes, Twitter (0.6 million documents)
3. **Direct Download** from AWS Publicdatasets, Kaggle & Stanford database (3.4 million documents)

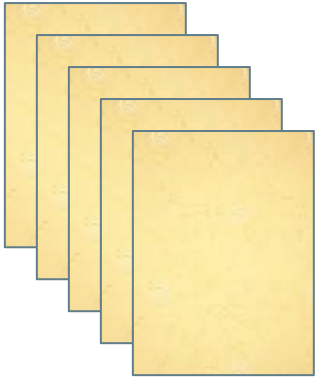
Search Algorithm

Step 1: Enter Search Statement

Step 2: Compute Document similarity with search string

Document_ID	Similarity_Score
1	0.1932
2	0.2513
3	0.2241
4	0.4391
5	0.2367
6	0.295
7	0.1814
8	0.0334
9	0.4633
10	0.0801
.	0.448
4 Million	0.2884

Step 3: Shows the 5 documents with the highest similarity score



Quantifying Similarity between 2 text documents

- We have used **Cosine Similarity** to measure the similarity between any 2 documents

- Define *A* and *B* be vectors of length *v*, where *v* is the number of unique words in your vocabulary.

- If *A* and *B* are of the form $[count(w_1), count(w_2), \ldots]$ where $w_1, w_2, \ldots, w_v \in V$ for two strings which are composed of elements of V, then the cosine similarity between *A* and *B* is defined as

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Python Implementation

```
#Function for measuring consine similarity
def cosine_sim(text1):
    """
    :param text1:
    :return: Cosine similarity based on TF-IDF score
    """
    vec1 = text_to_vector(text1.lower())
    vec2 = text_to_vector(search_text.lower())
    intersection = set(vec1.keys()) & set(vec2.keys())
    numerator = sum([vec1[x] * vec2[x] for x in intersection])
    sum1 = sum([vec1[x]**2 for x in vec1.keys()])
    sum2 = sum([vec2[x]**2 for x in vec2.keys()])
    denominator = math.sqrt(sum1) * math.sqrt(sum2)

    if not denominator:
        return 0.0
    else:
        return float(numerator) / denominator
```

Specifications:

1. Local: Ran on Macbook (8 GB RAM) without using any external module
2. Used 16 Node EC2 instance (320 Gibs) with "**multiprocessing**" library to do parallel processing on 16 partitions of dataset
3. Nearly 300% improvement in search speed in the distributed environment
4. 30 lines of code

Pyspark Implementation

```python
def get_unique_word_list(article_str, search_str):
    alist, slist = (article_str.split(), search_str.split())
    return list(set(alist + slist))

def get_word_count_vec(unique_word_list, str_to_search):
    counts = [str_to_search.lower().split().count(w) for w in unique_word_list]
    return np.array(counts)

def calc_cosine_sim(line, search_term):
    index, article = line[0], line[1]
    uniqs = get_unique_word_list(article, search_term)
    avec, svec = get_word_count_vec(uniqs, article), get_word_count_vec(uniqs, search_term)
    sim = 1 - cosine(avec, svec)
    return (sim, index)

test_search_str = 'Dog food'
d = master.filter(lambda line: check_words_in_text(line[1], test_search_str))
d.map(lambda line: calc_cosine_sim(line, test_search_str)).takeOrdered(5, key=lambda line: line[1])
```

Specifications:

1. Local: Ran on MacBook Pro (8GB RAM)
2. Distributed: Run on 2 Instance EMR (r3.4xlarge) using 16 partitions
3. On average a 6-fold decrease in run time is observed using EMR relative to local run times
4. 50 lines of code

Hive Implementation

```
insert overwrite table cosine_similarity
select lhs.docid,
sum(lhs.TFIDF_score * rhs.TFIDF_score)/
((sqrt(sum(lhs.TFIDF_score * lhs.TFIDF_score))*
(sqrt(sum(rhs.TFIDF_score * rhs.TFIDF_score))))*1.0
as cos_sim
from tfidf as lhs
inner join q_tfidf as rhs
on lhs.term = rhs.term
group by lhs.docid, rhs.docid
SORT BY cos_sim DESC
LIMIT 50;
```

Terms
doc_id BIGINT
terms STRING

TFIDF
doc_id BIGINT
term STRING
score FLOAT

Documents
doc_id BIGINT
txt array <STRING>

Term Frequency
doc_id BIGINT
term STRING
term_freq INT

Cosine Similarity
doc_id BIGINT
score FLOAT

Doc Frequency
doc_id BIGINT
doc_freq INT

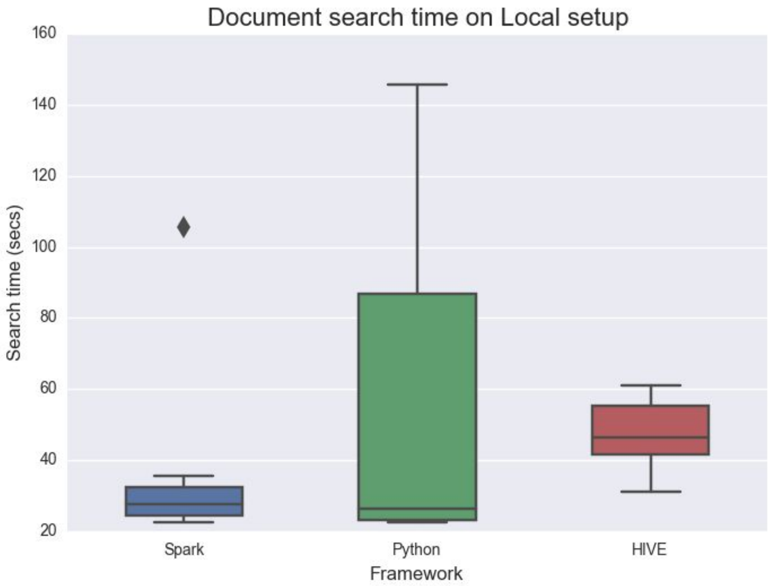
Specifications:

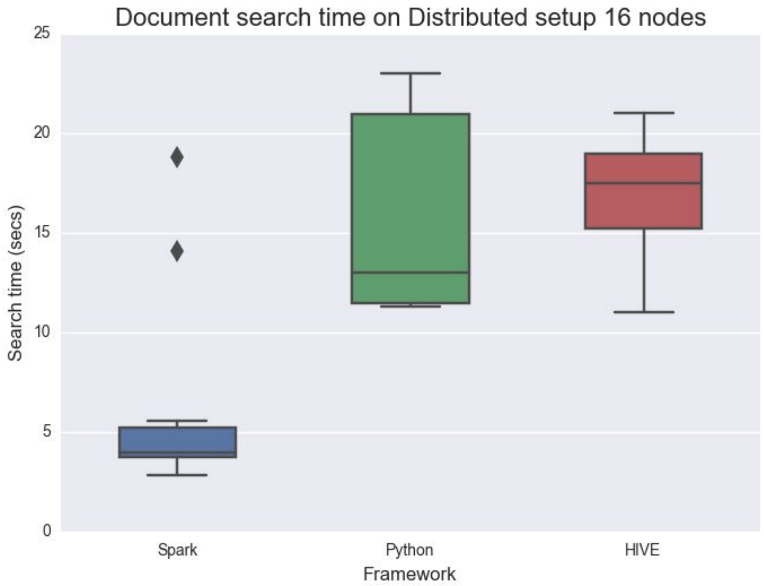
1. Local: Ran on MacBook Pro (8GB RAM)
2. Distributed implementation on a r3.4x large (320 Gibs) EMR Instance with 16 buckets
3. Search speed improved by 200% in distributed
4. 50 lines of code

Challenges:

1. TF-IDF computations was a challenge on large text corpus
2. Hive running locally on huge data set.

Search speed in the 3 frameworks





Observations:

1. **n.b** the Y-axis has changed from **180 seconds** (local) to **25 seconds** (distributed)
2. Spark has a median time of **4.2 seconds** in the distributed setup, hence it is our winning approach

Final Conclusions

1. Pyspark yields the best performance for our use case

2. Higher RAM & Partitioning boost search speed

3. NLP Techniques such as Latent Semantic Indexing can improve the quality of results

Performance Boosting Strategies

1. Database subsetting with the search string combinations, to get rid of all insignificant documents

2. Database partitioning and Multithreading also boost speed dramatically, enabling efficient RAM usage

Thank you

Project available on **Github**

UNIVERSITY OF
SAN FRANCISCO

Master of Science
in Analytics