직접 만들면서 이해하는 검색엔진의 작동 원리

김기동

대충 자기소개 들어갈 곳

이창희 (김기동 아님)

- Software Engineer @ Marpple
- 6년차 개발자
- Golang 좋아함
- PyCon KR 2019, JSConf KR 2022 발표자

 blurfx
 blurfx
 https://xo.dev

오늘 발표에서 다루지 않는 것

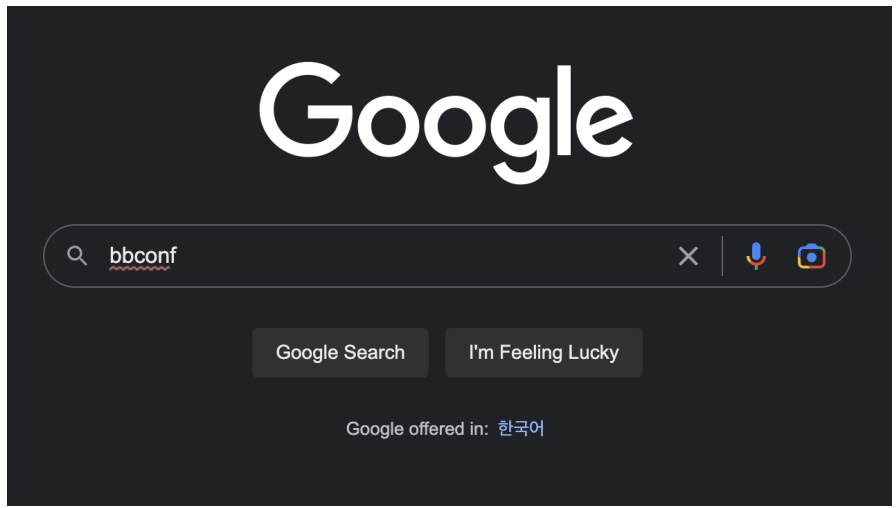
- 검색 퀄리티가 좋은 검색 엔진
- 형태소 분석
  - (당연하게도) CJK 지원
- 분산 처리
- ... 암튼 잘 만든 검색 엔진

오늘 발표에서 다루는 것

- 기본적인 검색 기능을 구현 하는방법
  - 문자열 검색
  - SQL
- 기초적인 방식의 한계점
- 검색 엔진 소프트웨어들
- Golang으로 만드는 단순한 검색 엔진

검색 엔진?

- 유저에게서 검색어(쿼리)를 입력받아

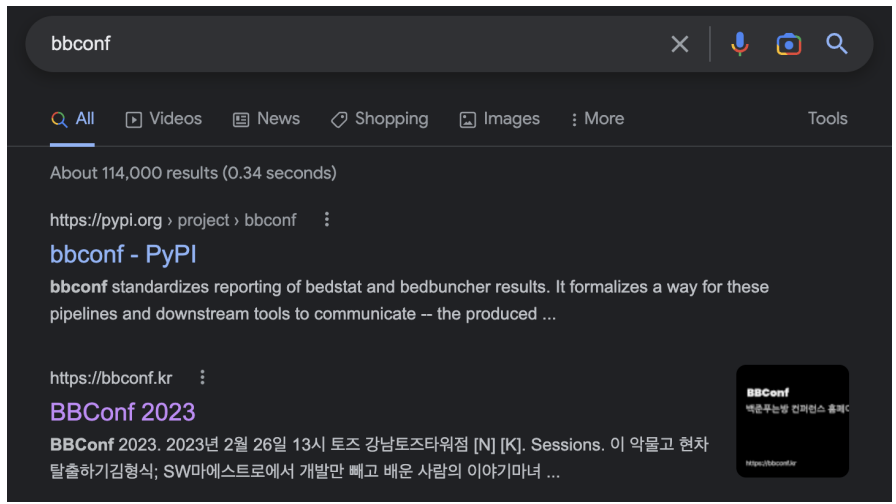


검색 엔진?

- 유저에게서 검색어(쿼리)를 입력받아
- 검색어에 가장 잘 맞다고 판단되는 문서들을 찾아서

검색 엔진?

- 유저에게서 검색어(쿼리)를 입력받아
- 검색어에 가장 잘 맞다고 판단되는 문서들을 찾아서
- 검색 결과를 보여주는 일을 하는 프로그램



간단한 검색 알고리즘

가장 간단한건 역시 Exact Matching

```
strings.Contains("document text", "query")
```

간단한 검색 알고리즘

가장 간단한건 역시 Exact Matching

```
var query string
var documents []Document

for _, doc := range documents {
  // 만약 쿼리가 문서 본문에 포함되어 있다면
  if strings.Contains(doc.Content, query) {
    // 뭔가 함.. like: 검색 결과에 추가
  }
}
```

간단한 검색 알고리즘

가장 간단한건 역시 Exact Matching

```
var query string
var documents []Document

for _, doc := range documents {
  // 만약 쿼리가 문서 본문에 포함되어 있다면
  if strings.Contains(doc.Content, query) {
    // 뭔가 함.. like: 검색 결과에 추가
  }
}
```

하지만 쿼리가 여러 단어로 구성되어 있다면? 🤔

간단한 검색 알고리즘: 예외

문서의 내용이 `"red apple and orange juice"`이고
검색 쿼리로 `"apple juice"`가 들어오는 경우를 상상해보자.

```
strings.Contains("red apple and orange juice", "apple juice")
```

- 검색 쿼리에 있는 `apple`과 `juice` 모두 문서에 포함되어 있다
- 그래서 이 경우는 검색 결과에 포함되어야 할 것 같다
- 하지만 위의 코드는 `false`를 반환한다

→ **검색 결과에 포함되지 않는다.**

간단한 검색 알고리즘: 예외

Idea: 쿼리를 단어 단위로 쪼개서 검사하자

빠르다고 취급되는 Knuth-Morris-Pratt의 시간 복잡도는 $O(N+M)$이다 (문서의 길이 N, 쿼리의 길이 M)

만약에 쿼리가 여러개라면?

Q개의 쿼리가 있고, 모든 쿼리의 길이 총 합이 S라 해보자

KMP에서 전처리를 하는데 $O(S)$의 시간이 걸리고 검색을 하는데 $O(QM)$의 시간이 걸리므로 총 $O(S+QM)$의 시간이 걸린다.

문서도 여러개라면?

DB로만 검색 기능 구현하기

앞에서 간단한 문자열 검색으로 문서 검색을 어떻게 할까에 대해 고민해봤지만, 실제로 직접 구현하는 경우는 드물다

Why

보통 검색할 문서들은 데이터베이스에 저장되어 있고, 데이터베이스 기능과 SQL을 사용하면 편하게 처리할 수 있다

```
-- LIKE
SELECT ... FROM documents WHERE text LIKE '%apple juice%'

-- REGEXP
SELECT ... FROM documents WHERE text REGEXP '.*apple juice.*'

-- FULLTEXT INDEX + MATCH .. AGAINST
CREATE TABLE documents ... text TEXT, FULLTEXT(text)
SELECT ... FROM documents WHERE MATCH(text) AGAINST('apple juice' IN BOOLEAN MODE)
```

LIKE, REGEXP 검색은 문서 수와 텍스트 양이 많아질수록 성능이 가시적으로 떨어지지만
FULLTEXT 인덱스를 사용한 검색은 비교적 성능이 좋다

하지만 DB만으로는 안된다

DB의 검색 기능은 *검색을 한다*는 목적으로는 충분하지만, *검색을 잘 한다*와는 거리가 멀다

- 약간의 오타가 있어도 검색 결과에 포함되지 않는다
- 비정형 데이터를 처리하기 힘들다
- FULLTEXT 인덱스를 사용해도 문서가 많아지면 결국 성능에 한계가 온다
- 검색 결과의 품질을 보장하지 못한다
- 검색 결과 선정에 개입이 불가능하다
- ...

검색 엔진 소프트웨어

- Apache Lucene
  - Apache Solr
  - Elasticsearch
  - OpenSearch
  - MongoDB Atlas

모두 Lucene을 기반으로 만들어졌다

Lucene은 검색 엔진을 구현하기 위한 기능을 제공해주는 저수준 라이브러리

매우 빠른 성능을 자랑하지만 라이브러리인만큼 기능이 풍부하지는 않다
다른 소프트웨어들은 Lucene을 기반으로 여러 기능을 붙여 고수준으로 만든 것

→ **이 검색 엔진들은 어떻게 해서 검색이 빠른걸까? 직접 만들면서 알아보자**

문서에 이 단어가 포함되어 있는지 알고 싶다

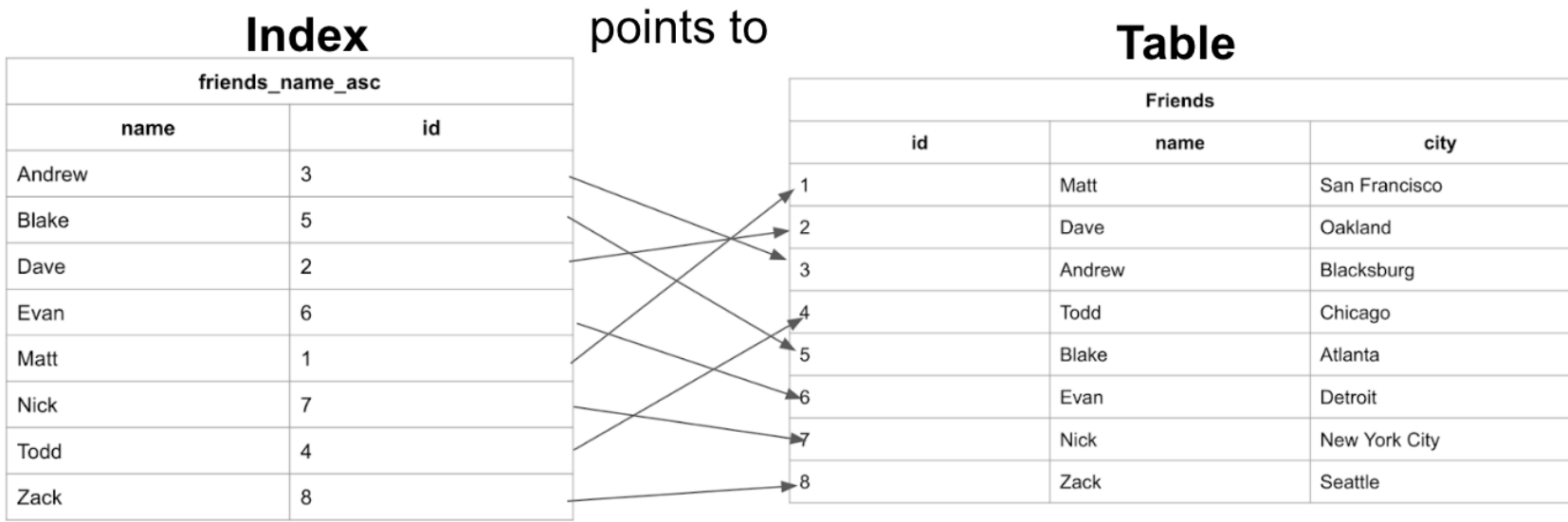
유저가 입력하는 쿼리는 하나의 단어가 아닌 긴 문장이 될수도 있다

앞에서 했던 것처럼 단순히 단어 단위로 잘라 포함 여부를 검사하면 매우 느릴것

그렇다면 어떻게...?

Index

데이터베이스에서 빠르게 데이터를 찾기 위해 같은 키를 가진 데이터를 모아놓은 자료구조



friends_name_asc	
name	**id**
Andrew	3
Blake	5
Dave	2
Evan	6
Matt	1
Nick	7
Todd	4
Zack	8

Friends		
id	**name**	**city**
1	Matt	San Francisco
2	Dave	Oakland
3	Andrew	Blacksburg
4	Todd	Chicago
5	Blake	Atlanta
6	Evan	Detroit
7	Nick	New York City
8	Zack	Seattle

하지만 텍스트 검색으로 쓰는 경우에는 키가 너무 많다

Inverted Index

문서를 단어 단위로 쪼개서, 각 단어가 어떤 문서에 포함되어 있는지를 저장하는 자료구조

Forward Index가 각 문서가 어떤 값을 포함하고 있는지를 저장하는 반면,
Inverted Index는 임의의 값이 어떤 문서에 포함되어 있는지를 저장한다

DOCUMENT ID	TEXT
doc1	apple favored chocolate
doc2	orange juice
doc3	apple orange juice

TERM	DOCUMENT ID
apple	doc1, doc3
chocolate	doc1
favored	doc1
juice	doc2, doc3
orange	doc2, doc3

Inverted Index: 구현

```go
type Document struct {
    ID      int
    Content string
}

type InvertedIndex map[string][]int // Map<string, Array<int>>

func BuildInvertedIndex(documents []Document) InvertedIndex {
    index := make(InvertedIndex)

    for _, doc := range documents {
        tokens := strings.Fields(doc.Content)

        for _, token := range tokens {
            if _, ok := index[token]; !ok {
                index[token] = make([]int, 0)
            }
            index[token] = append(index[token], doc.ID)
        }
    }

    return index
}
```

Inverted Index: 구현

```
func main() {
    documents := []Document{
        {ID: 0, Content: "apple favored chocolate"},
        {ID: 1, Content: "orange juice with candy"},
        {ID: 2, Content: "apple orange juice"},
    }

    index := BuildInvertedIndex(documents)
    for key, value := range index {
        fmt.Println(key, value)
    }
}
```

KEY	DOCUMENT ID
apple	0, 2
candy	1
favored	0
chocolate	0
orange	1, 2
juice	1, 2
with	1

이제 쿼리에 `N`개 단어가 있을때, 해당 단어들이 포함된 문서를 `O(N)` 시간에 찾을 수 있다!

문서를 찾는게 끝이 아니다

이제 우리는 주어진 쿼리로 원하는 문서를 빠르게 찾을 수 있다

하지만, 검색 엔진에서 중요한건 검색 결과이다

`apple juice candy`라는 쿼리가 있을때, `apple juice candy` 포함하는 문서를 찾아도

그 문서 사이에서 우선순위를 정해주지 않으면 품질이 좋은 검색 결과라 보기 힘들 것이다

그렇다면 어떻게 우선순위를 정할까?

문서 우선순위 정하기

가장 단순한 방법을 생각해보면 매칭된 단어의 개수를 세는 것

```
func main() {
    documents := []Document{
        {ID: 0, Content: "apple favored chocolate"},
        {ID: 1, Content: "orange juice with candy"},
        {ID: 2, Content: "apple orange juice"},
    }

    index := BuildInvertedIndex(documents)
    query := "apple juice candy"
    tokens := strings.Fields(query)
    matchCount := make(map[int]int)
    for _, token := range tokens {
        if docs, ok := index[token]; ok {
            for _, docId := range docs {
                matchCount[docId]++
            }
        }
    }

    // matchCount를 정렬한 뒤 문서 찾아서 반환하기
}
```

문서 우선순위 정하기

나쁘지는 않지만 문서 검색에서는 더 고려해야 할 것이 있다

바로 해당 단어가 문서에서 얼마나 중요한지를 고려해야 한다

예를 들어, `apple juice candy`라는 쿼리가 있을때, `apple juice candy` 포함하는 문서를 찾아도

`apple`이 `juice`보다 더 중요한 단어일 수 있다

그렇다면 어떻게 단어의 중요도를 정할까?

TF-IDF

Term Frequency - Inverse Document Frequency

단어의 중요도를 정하는 방법은 여러가지가 있지만 가장 널리 쓰임

여러 문서에서 자주 등장하는 단어는 중요도가 낮고, 특정 문서에서만 자주 등장하는 단어는 중요도가 높다고 판단하는 방법

$$tfidf = tf(d, t) \times idf(t)$$

TF-IDF: Term Frequency

$tf(d, t)$ = 문서 d에서 단어 t가 등장한 횟수

DOCUMENT ID	TEXT
doc1	apple favored chocolate
doc2	orange juice
doc3	apple orange juice

→

TERM	DOCUMENT ID	TERM FREQUENCY
apple	doc1, doc3	2
chocolate	doc1	1
favored	doc1	1
juice	doc2, doc3	2
orange	doc2, doc3	2

TF-IDF: Term Frequency: 구현

Term Frequency 구현은 단순히 해당 단어가 문서에서 몇번 등장하는지를 세면 된다

```
tf := strings.Count(se.documents[docID].Content, token)
```

TF-IDF: Inverse Document Frequency

Inverse Document Frequency는 단어가 각 문서에 등장하는 빈도(TF)와
전체 문서에서 단어가 등장하는 빈도(DF)를 고려하여 단어의 중요도를 계산하는 방법을 말한다

전체 문서 n개가 있고, 단어 t를 찾을 때:

$df(t) =$ 단어 t가 등장한 문서의 수

$idf(t) = log(\frac{n}{1+df(t)})$

TF-IDF: Inverse Document Frequency: 구현

Inverse Document Frequency는 다음과 같이 구할 수 있다

```
tokens := strings.Fields(query)
scores := make(map[int]float64)

for _, token := range tokens {
    if matchedDocuments, ok := index[token]; ok {
        // Log(전체 문서 수 / 단어 token이 등장한 문서 수)
        idf := math.Log(float64(len(documents)) / float64(len(matchedDocuments)))
    // ...
    }
}
```

TF-IDF: Inverse Document Frequency: 구현

DOCUMENT ID	TEXT
doc1	apple favored chocolate
doc2	orange juice with candy
doc3	apple orange juice
doc4	banana juice
doc5	hey hey hey hey

TERM	DOCUMENT ID	TERM FREQUENCY	DOCUMENT FREQUENCY	INVERSE DOCUMENT FREQUENCY
apple	doc1, doc3	2	2	0.405
banana	doc4	1	1	0.693
candy	doc2	1	1	0.693
chocolate	doc1	1	1	0.693
favored	doc1	1	1	0.693
hey	doc5	4	1	0.693
juice	doc2, doc3, doc4	3	3	0.287
orange	doc2, doc3	2	2	0.405
with	doc2	1	1	0.693

TF-IDF: 구현

TF와 IDF를 구하는 방법을 알았으니 $tfidf = tf(d, t) \times idf(t)$을 따라서 구현하면 된다

```
tokens := strings.Fields(query)
scores := make(map[int]float64)

for _, token := range tokens {
    if matchedDocuments, ok := index[token]; ok {
        // Log(전체 문서 수 / 단어 token이 등장한 문서 수)
        idf := math.Log(float64(len(documents)) / float64(len(matchedDocuments)))
        for _, docId := range matchedDocuments {
            tf := float64(strings.Count(documents[docId].Content, token))
            scores[docId] += tf * idf
        }
    }
}
```

TF-IDF 데모

BM25

Best Matching 25 : TF-IDF를 더 개선한 알고리즘

무엇이 개선 되었나?

TF-IDF 식을 변형하여, 문서의 길이가 길어질수록 단어의 중요도가 떨어지도록 한다.

Q는 쿼리(단어)의 집합으로 Q_1, Q_2, ..., Q_n의 쿼리들이 있다.

k_1과 k_2는 양수 상수이다. 이 값을 튜닝하여 스코어를 조절할 수 있다.

$|D|$는 문서 D의 길이이고, avgdl은 전체 문서의 평균 길이이다.

$$\mathrm{BM25}(D, Q) = \sum_{i=1}^{n} \mathrm{idf}(t_i) \cdot \frac{f_{i,Q} \cdot (k_1 + 1)}{f_{i,Q} + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\mathrm{avgdl}})} \cdot \frac{(k_2 + 1) \cdot f_{i,D}}{k_2 + f_{i,D}}$$

문서 길이를 점수 계산식의 분모로 취해 문서의 길이가 길어질수록 단어의 중요도가 떨어진다.

IDF 함수를 통해 다른 문서에서 잘 등장하지 않는 단어는 더 중요하다고 판단한다.

BM25 데모

더 신경 써야 할 것

- 불용어 제거
  - 문법적으로는 의미가 있지만, 검색에서는 의미가 없는 단어들
  - ex) the, a, an, is, are, was, were, be, been, am, will, would, could, should, may, might, must, do, does, did, have, has, had, ...
- 특수문자 제거
- 오타 교정
  - 오타를 교정하기 보다 비슷한 단어를 찾는 편이 쉬움
  - Fuzzy Search로 Damerau–Levenshtein Distance를 많이 사용
- 형태소 분석
- 복수형, 과거형, 현재형
- 벡터 검색
  - ex) 댕댕이 -> 강아지
  - ex) 피리부는 대머리 -> 주호민