FastCosine:
Approximated Neighbor Search for Sparse Vector

Hyunjoong Kim

soy.lovit@gmail.com

https://github.com/lovit/fastcosine

Retrieval System as Neighbor Search

- 문서는 두 가지 방식으로 DB에 저장됨
  - Sparse representation: TFIDF, Keyword, …
  - Distributed representation: Doc2Vec, Sentence embedding, …

- 두 가지 방식은 서로 다른 장점이 있음
  - Sparse representation: 어떤 단어들이 포함하였는지 해석이 용이하며, 문서의 길이가 짧을 경우 적은 용량
  - Distributed representation: semantic search가 가능하지만, 모든 문서가 동일한 embedding vector 크기의 용량이 필요 (단문의 경우 단어 수 보다 큰 공간 필요)

Sparse representation은 여전히 좋은 방법

- Document classification에서 Bag-of-Words 모델은 일반적인 경우에 distributed representation 방법과 큰 성능 차이가 나지는 않음

Model	AG	Sogou	DBP	Yelp P.	Yelp F.	Yah. A.	Amz. F.	Amz. P.
BoW (Zhang et al., 2015)	88.8	92.9	96.6	92.2	58.0	68.9	54.6	90.4
ngrams (Zhang et al., 2015)	92.0	97.1	98.6	95.6	56.3	68.5	54.3	92.0
ngrams TFIDF (Zhang et al., 2015)	92.4	97.2	98.7	95.4	54.8	68.5	52.4	91.5
char-CNN (Zhang and LeCun, 2015)	87.2	95.1	98.3	94.7	62.0	71.2	59.5	94.5
char-CRNN (Xiao and Cho, 2016)	91.4	95.2	98.6	94.5	61.8	71.7	59.2	94.1
VDCNN (Conneau et al., 2016)	91.3	96.8	98.7	95.7	64.7	73.4	63.0	95.7
`fastText`, $h = 10$	91.5	93.9	98.1	93.8	60.4	72.0	55.8	91.2
`fastText`, $h = 10$, bigram	92.5	96.8	98.6	95.7	63.9	72.3	60.2	94.6

Table 1: Test accuracy [%] on sentiment datasets. `FastText` has been run with the same parameters for all the datasets. It has 10 hidden units and we evaluate it with and without bigrams. For char-CNN, we show the best reported numbers without data augmentation.

Joulin, A., Grave, E., Bojanowski, P., & Mikolov, T. (2016). Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.

Sparse representation은 여전히 좋은 방법

- Information retrieval 분야에서 distributed representation이 이용되는 이유
  - Term weighting, language modeling smoothing
  - Query expanding

- 위 기술들이 필요할 때 distributed representation을 이용해야 효과가 있음
  - Query expansion 역시 fully distributed representation을 이용하기보다 sparse representation과 혼합하여 이용하는 경우도 많음

Mitra, B., & Craswell, N. (2017, February). Neural Text Embeddings for Information Retrieval. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining* (pp. 813-814). ACM.

Sparse representation은 여전히 좋은 방법

- Palangi et al., (2016)은 LSTM 기반 sentence embedding을 이용하여 retrieval engine을 만듦

- Normalized Discounted Cumulative Gain (NDCG @k) 기준, TFIDF + Cosine인 BM25의 30.8%와 약 2.5% 성능 차이

$$DCG_k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

$nDCG_K = \frac{DCG_k}{IDCG_k}$, $IDCG_k$ is maximum possible DCG

rel_i : relevance score within [0, 1]

Model	NDCG @1	NDCG @3	NDCG @10
Skip-Thought off-the-shelf	26.9%	29.7%	36.2%
Doc2Vec	29.1%	31.8%	38.4%
ULM	30.4%	32.7%	38.5%
BM25	30.5%	32.8%	38.8%
PLSA (T=500)	30.8%	33.7%	40.2%
DSSM (nhid = 288/96) 2 Layers	31.0%	34.4%	41.7%
CLSM (nhid = 288/96, win=1) 2 Layers, 14.4 M parameters	31.8%	35.1%	42.6%
CLSM (nhid = 288/96, win=3) 2 Layers, 43.2 M parameters	32.1%	35.2%	42.7%
CLSM (nhid = 288/96, win=5) 2 Layers, 72 M parameters	32.0%	35.2%	42.6%
RNN (nhid = 288) 1 Layer	31.7%	35.0%	42.3%
LSTM-RNN (ncell = 32) 1 Layer, 4.8 M parameters	31.9%	35.5%	42.7%
LSTM-RNN (ncell = 64) 1 Layer, 9.6 M parameters	32.9%	36.3%	43.4%
LSTM-RNN (ncell = 96) 1 Layer, n = 2	32.6%	36.0%	43.4%
LSTM-RNN (ncell = 96) 1 Layer, n = 4	33.1%	36.5%	43.6%
LSTM-RNN (ncell = 96) 1 Layer, n = 6	33.1%	36.6%	43.6%
LSTM-RNN (ncell = 96) 1 Layer, n = 8	**33.1%**	**36.4%**	**43.7%**
Bidirectional LSTM-RNN (ncell = 96), 1 Layer	**33.2%**	**36.6%**	**43.6%**

* Palangi, H., Deng, L., Shen, Y., Gao, J., He, X., Chen, J., ... & Ward, R. (2016). Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, *24*(4), 694-707.
* https://www.kaggle.com/wiki/NormalizedDiscountedCumulativeGain

Conclusion

Sparse representation은 나쁘지 않다

Information Retrieval은 Sparse/distributed representation 이던,
결국 **neighbor search** 문제를 풀어야 한다

Locality Sensitive Hashing (LSH)

- LSH는 retrieval quality를 높이기 위하여 Hashing Tables을 더 많이 중첩하는, randomness를 이용하는 방법을 이용.

  → tables을 더 많이 쌓아도 성능 향상은 거의 일어나지 않으며, 비용만 많이 들어감

- Query time 안에 계산이 되어야 했기 때문에, (속도, 품질)을 모두 잡는 확실한 인덱서가 필요

Model concept

- 데이터가 매우 sparse 할 때, Cosine similarity로 유사도가 정의된다면, inverted index를 이용하여 *k*-nearest neighbors를 효율적으로 찾을 수 있다.

  - Cosine은 term weight의 곱으로 계산되기 때문에, query terms 중 하나 이상의 terms을 지닌 문서만이 'similarity > 0' 이며, sparse data이기 때문에 해당 terms을 지닌 문서의 숫자는 적기 때문이다.
  - 그러므로 term 기준으로 indexing이 되어있는 inverted indexer가 적절한 방식이다

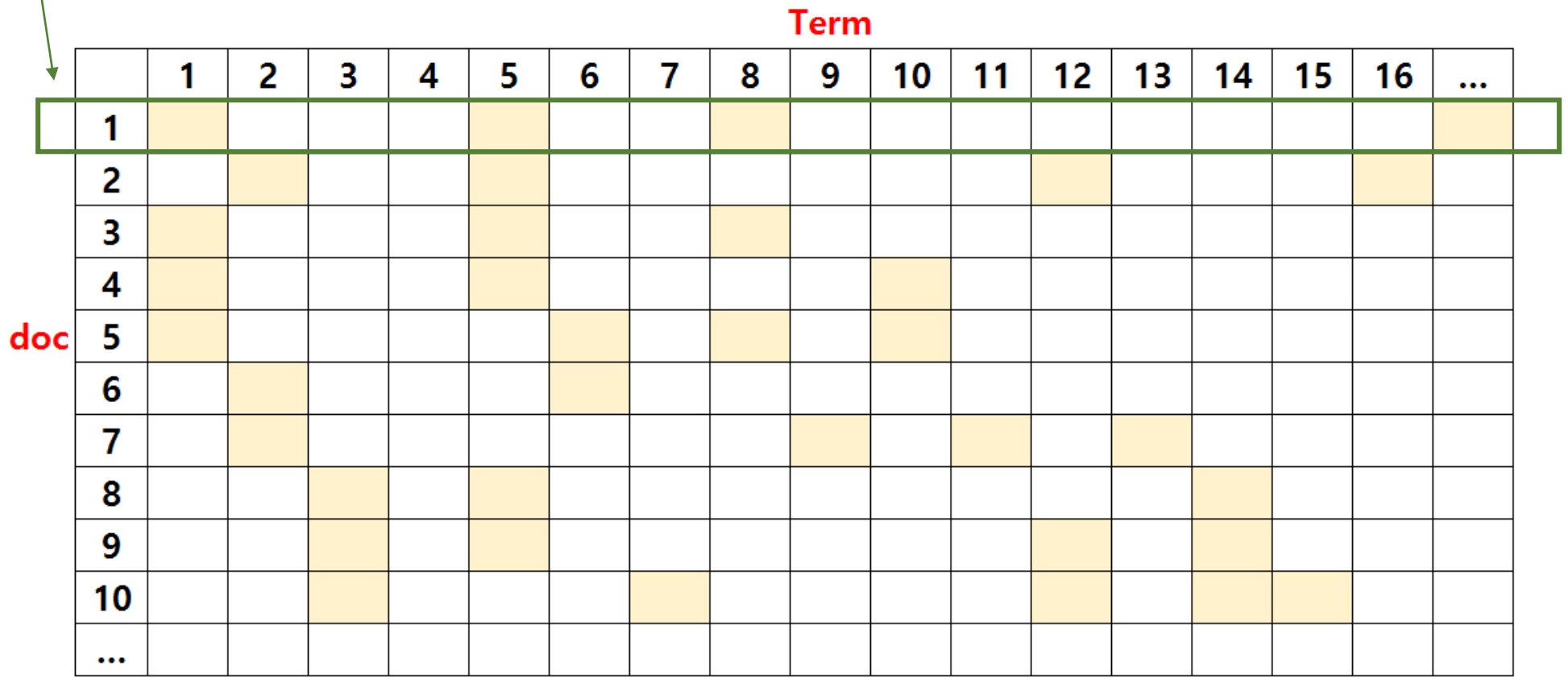
{
'$term_1$': [(doc_1, $w_{1,1}$), (doc_{101}, $w_{101,1}$), …],
'$term_2$': [(doc_{81}, $w_{81,2}$), (doc_{275}, $w_{275,2}$), …],
'$term_3$': [(doc_{27}, $w_{27,3}$), (doc_{39}, $w_{39,3}$), …],
…
}

inverted indexer

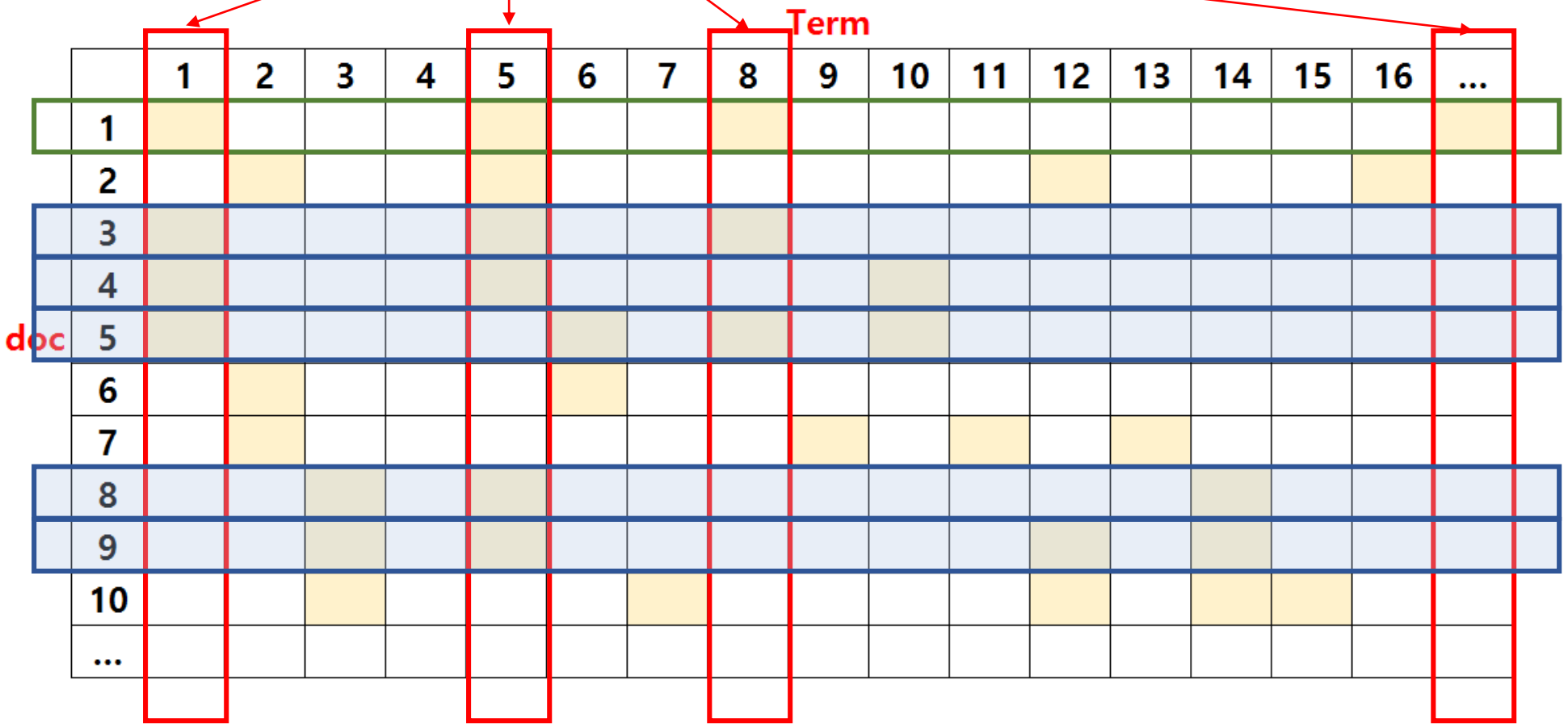
Term

doc	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
1	■				■			■									■
2		■			■							■				■	
3	■				■			■									
4	■				■					■							
5	■					■		■		■							
6		■				■											
7		■							■		■		■				
8			■		■									■			
9			■		■							■		■			
10			■				■					■		■	■		
...																	

Which documents are similar with doc 1?



The similar documents must have at least one common term



Model concept

- Inverted indexer를 이용하여 Cosine을 계산할 때에는 doc 기준이 아닌 term 기준으로 iteration을 돌아야 함

Vanilla pseudo code

```
def find_similar_docs(query):
    scores = { }
    for t in query:
        for d in get_doc_having(term):
            scores[doc] += w(d, t) * w(q, t)
    similars = sort_by_value(scores)
    Return similars
```

Model concept

- If query and documents are L2 normalized

$$\cos(q,d) = \sum_{t \in q} w(q,t) * w(d,t)$$

- When we split query terms to two parts T_1 and T_2,

$$\cos(q,d) = \sum_{t \in T_1} w(q,t) * w(d,t) + \sum_{t` \in T_2} w(q,t`) * w(d,t`)$$

Model concept

$$\cos(q,d) = \sum_{t \in T_1} w(q,t) * w(d,t) + \sum_{t` \in T_2} w(q,t`) * w(d,t`)$$

- T_1의 모든 terms을 확인하였다면,

  T_2에서 scores[doc]의 최대 증가분은 $\max_{t` \in T_2} w(q,t`)$

- 충분히 탐색을 했다고 판단되면, 어느 순간

  for t in query를 멈추면 됨 (earlystop)

Vanilla pseudo code

```
def find_similar_docs(query):
    scores = { }
    for t in query:
        for d in get_doc_having(term):
            scores[doc] += w(d, t) * w(q, t)
    similars = sort_by_value(scores)
    Return similars
```

Model concept

$$\cos(q,d) = \sum_{t \in T_1} w(q,t) * w(d,t) + \sum_{t` \in T_2} w(q,t`) * w(d,t`)$$

- 어떤 **query term order**로 탐색할 것인가?
- **어떤 문서를 top k의 후보**에 넣을 것인가?
- **언제 term loop을 멈출** 것인가?

Vanilla pseudo code

```
def find_similar_docs(query):
    scores = { }
    for t in query:
        for d in get_doc_having(term):
            scores[doc] += w(d, t) * w(q, t)
    similars = sort_by_value(scores)
    Return similars
```

Indexing

- FastCosine은 각 단어 (term)에 대하여 세 종류의 정보를 나눠서 저장한다
  - Weight (w): 단어 t에 대한 각 문서 d의 weight, w(t, d)
  - $|D_{tw}|$: 단어 t에 대해 weight의 값이 w인 문서의 개수
  - d_{twi}: 단어 t에 대하여 weight의 값이 w인 문서의 아이디

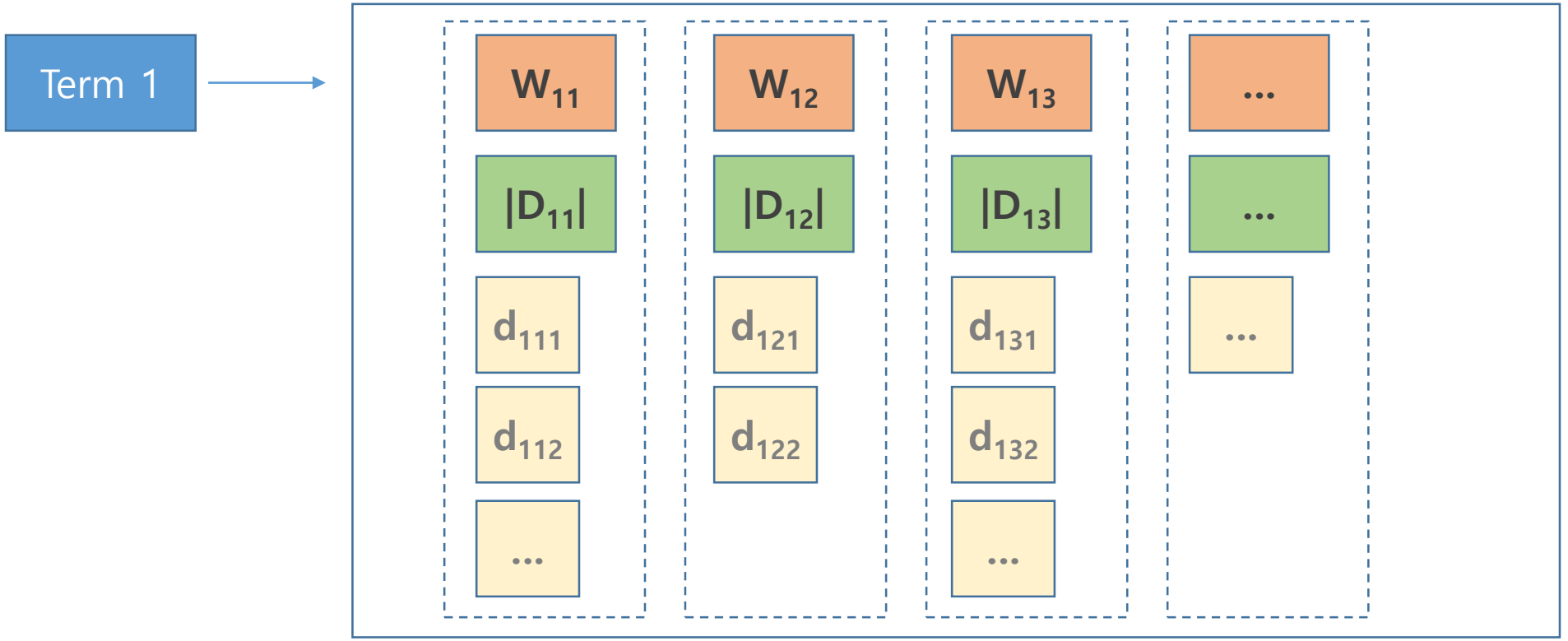
'term 1': {

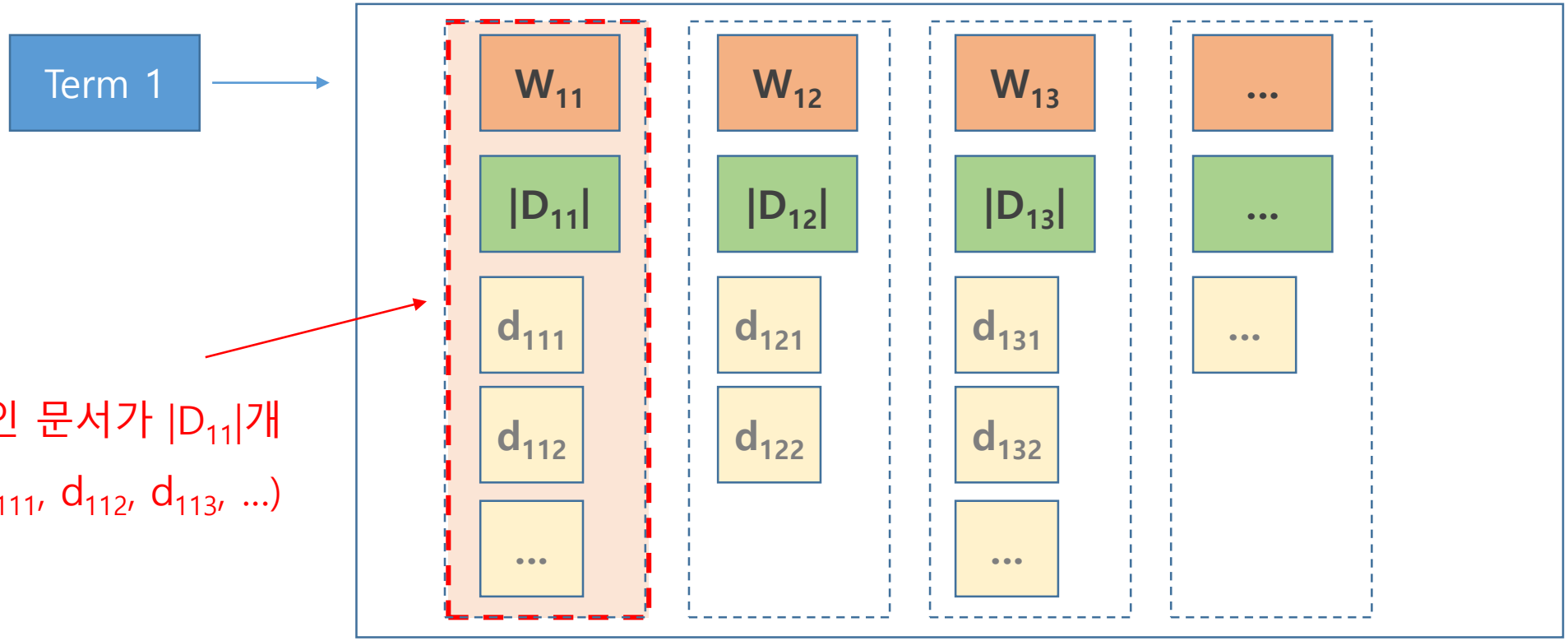
(w_{11}, w_{12}, w_{13}, ...),

($|D_{11}|$, $|D_{12}|$, $|D_{13}|$, ...),

((d_{111}, d_{112}, d_{113}), (d_{121}, d_{122}), (d_{131}, d_{132}, ...), ...)

}


Term 1
W_{11}
W_{12}
W_{13}
...
$|D_{11}|$
$|D_{12}|$
$|D_{13}|$
...
d_{111}
d_{121}
d_{131}
...
d_{112}
d_{122}
d_{132}
...
Term 2
W_{21}
W_{22}
W_{23}
...
$|D_{21}|$
$|D_{22}|$
$|D_{23}|$
...
d_{211}
d_{221}
d_{231}
...



$w(t_1) = w_{11}$인 문서가 $|D_{11}|$개

아이디는 (d_{111}, d_{112}, d_{113}, ...)

(w_{11}, $|D_{11}|$, (d_{111}, d_{112}, d_{113}, ...) 형식은 동일한 값 w11을 여러번 저장하지 않기 때문에

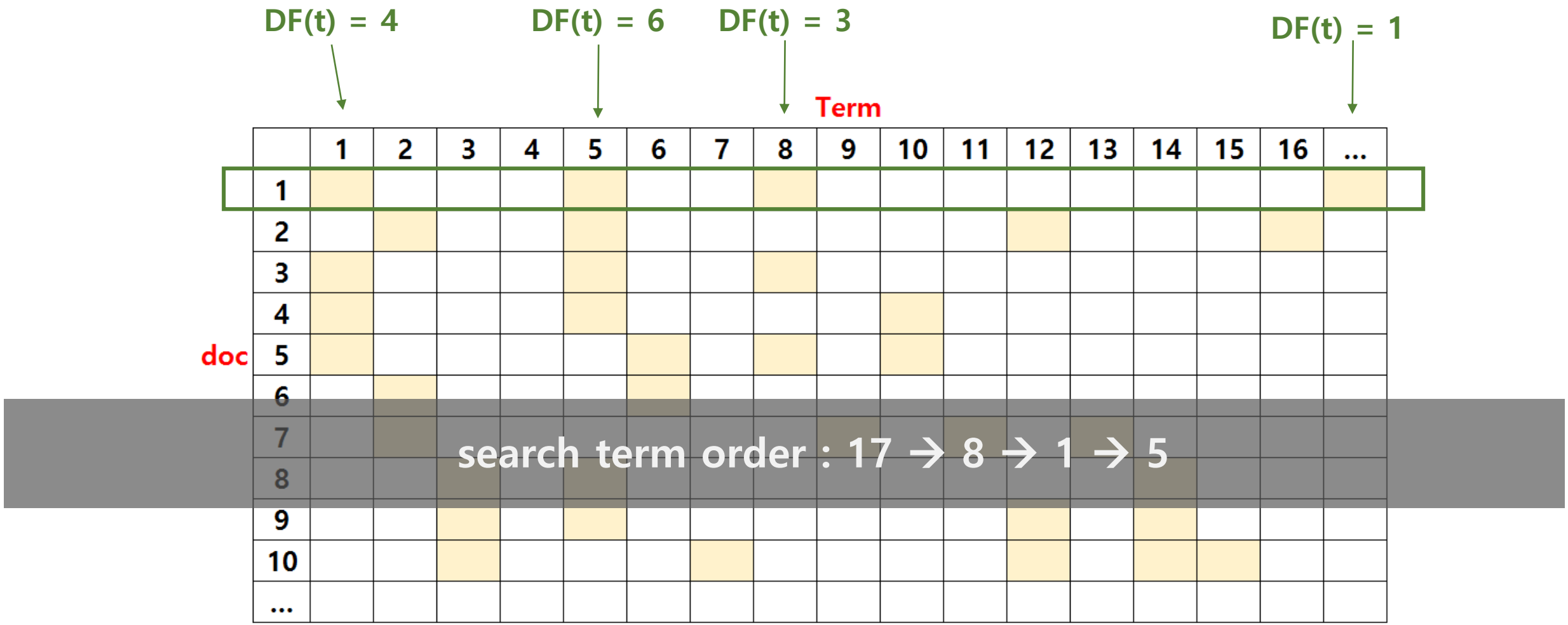
공간 효율적일 뿐더러, 동일한 weight를 지니는 문서가 몇 개 있는지 미리 확인할 수 있음

Query Processing: term order

$$\cos(q,d) = \sum_{t \in T_1} w(q,t) * w(d,t) + \sum_{t` \in T_2} w(q,t`) * w(d,t`)$$

```
def find_similar_docs(query):
    scores = { }
    for t in query:
        for d in get_doc_having(term):
            scores[doc] += w(d, t) * w(q, t)
    similars = sort_by_value(scores)
    Return similars
```

- **어떤 query term**부터 탐색할 것인가?
  - $w(q,t)$가 큰 term이 $\cos(q,d)$를 크게 만들 수 있다
  - $idf(t)$가 큰 term은 모든 문서의 유사도를 공평하게 높일 가능성이 높다.
  - $\boldsymbol{w(q,t) * idf(t)}$ **순서**대로 query term loop를 타자
    (Query에 대한 TF-IDF(t) scheme)


DF(t) = 4
DF(t) = 6
DF(t) = 3
DF(t) = 1
Term
doc
search term order : 17 → 8 → 1 → 5

Query Processing: filtering by “w(d, t) / max(w(d, t)”

$$\cos(q,d) = \sum_{t \in T_1} w(q,t) * w(d,t) + \sum_{t` \in T_2} w(q,t`) * w(d,t`)$$

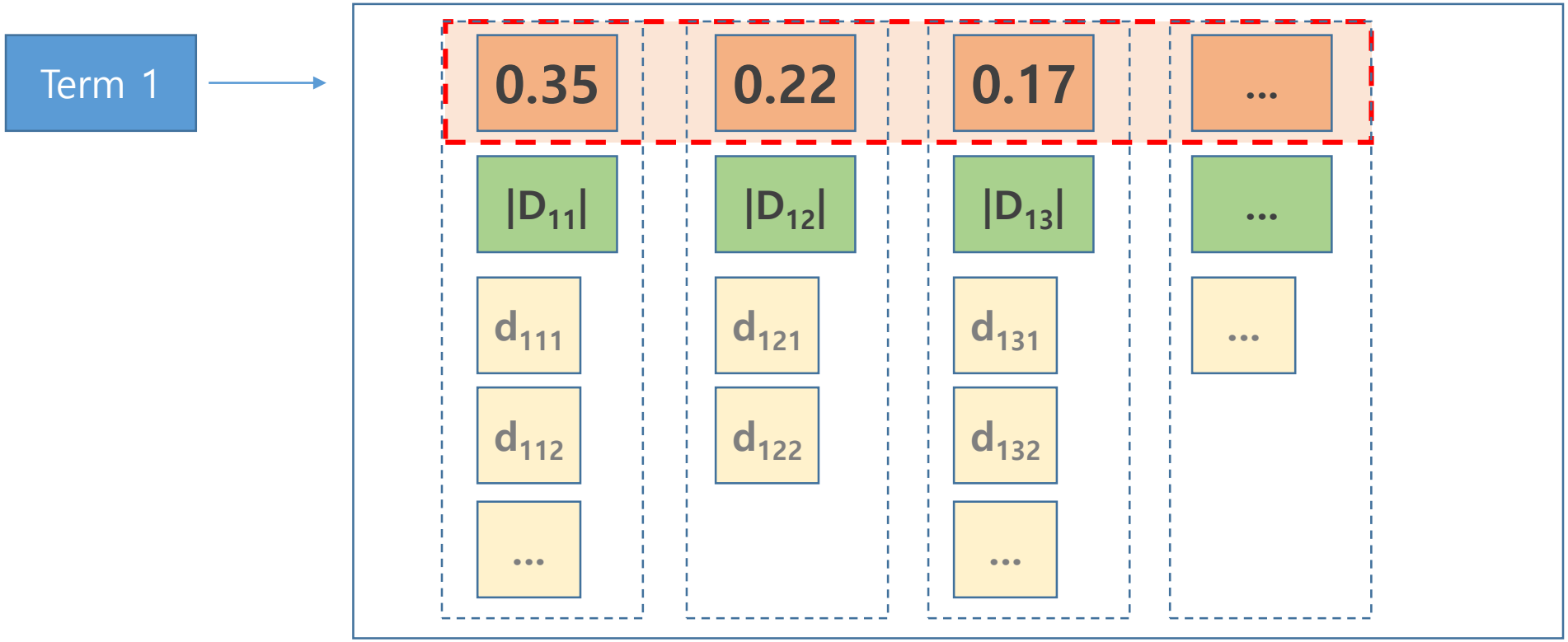
- **어떤 문서를 top k의 후보**에 넣을 것인가?

```
def find_similar_docs(query):
    scores = { }
    for t in query:
        for d in get_doc_having(term):
            scores[doc] += w(d, t) * w(q, t)
    similars = sort_by_value(scores)
    Return similars
```

$$\cos(q,d) = \sum_{t \in T, d \in D_t^+} w(q,t) * \boldsymbol{w(d,t)} + \sum_{t \in T, d \in D_t^-} w(q,t) * \boldsymbol{w(d,t)}$$

(w_{11}, w_{12}, ...)는 decreasing order로 정렬 후 저장,

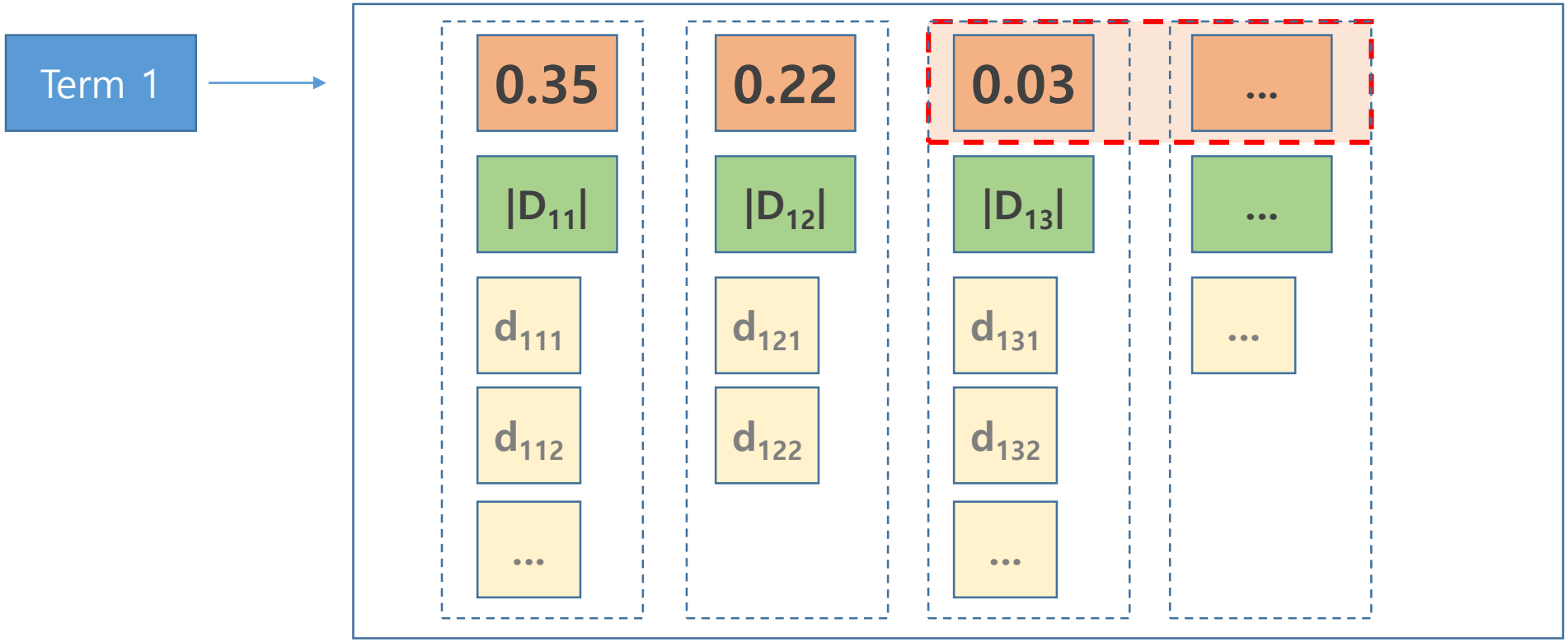
유사도에 영향을 많이 주는 문서들부터 고려할 수 있음



$$\cos(q,d) = \sum_{t \in T} w(q,t) * \boldsymbol{w(d,t)}$$

$w(q,t)$는 고정이기 때문에, $w(d,t)$가 큰 d가 유사도가 높음

다른 문서들보다 weight가 급격히 낮아지는 지점부터는 아예
k nearest neighbor 후보로 계산조차 하지 않아도 됨



$$\cos(q,d) = \sum_{t \in T, d \in D_t^+} w(q,t) * \boldsymbol{w(d,t)} + \sum_{t \in T, d \in D_t^-} w(q,t) * \boldsymbol{w(d,t)}$$

$w(d,t)$가 max $(w(d,t))$ * factor 이하인 지점부터는 무시

Query Processing: filtering by "*k* * factor"

$$\cos(q,d) = \sum_{t \in T_1} w(q,t) * w(d,t) + \sum_{t` \in T_2} w(q,t`) * w(d,t`)$$

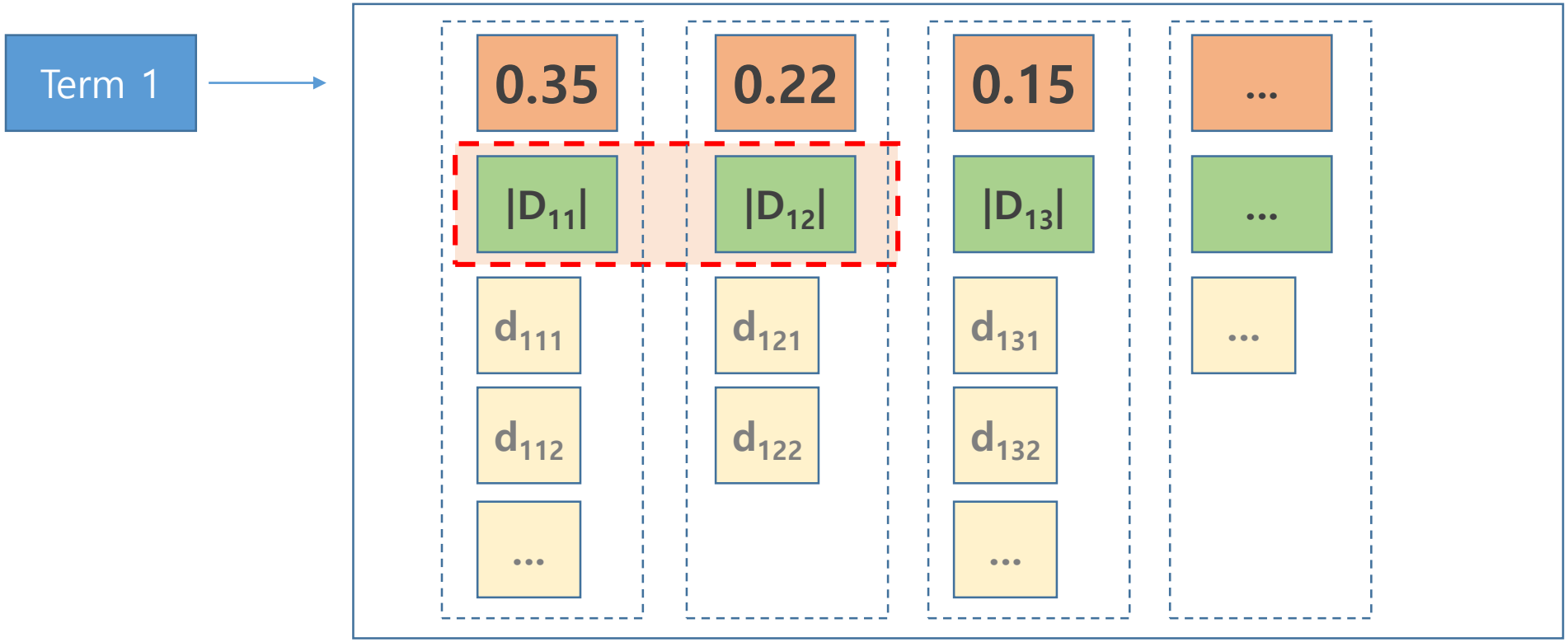
- **어떤 문서를 top k의 후보**에 넣을 것인가?

```
def find_similar_docs(query):
    scores = { }
    for t in query:
        for d in get_doc_having(term):
            scores[doc] += w(d, t) * w(q, t)
    similars = sort_by_value(scores)
    Return similars
```

$$\cos(q,d) = \sum_{t \in T, d \in D_t^+} w(q,t) * \boldsymbol{w(d,t)} + \sum_{t \in T, d \in D_t^-} w(q,t) * \boldsymbol{w(d,t)}$$

$|D_{11}| + |D_{12}|$의 크기가 충분히 크다면, 다른 문서들은 계산하지 않는다

scores[doc] += w(d, t) * w(q, t)의 횟수를 유지하도록 하는 역할



$$\cos(q,d) = \sum_{t \in T, \boldsymbol{d} \in \boldsymbol{D_t^+}} w(q,t) * \boldsymbol{w(d,t)} + \sum_{t \in T, d \in D_t^-} w(q,t) * \boldsymbol{w(d,t)}$$

D_t^+의 개수는 k-NN의 k * factor 이하가 되도록 제한

Query Processing: Early stop

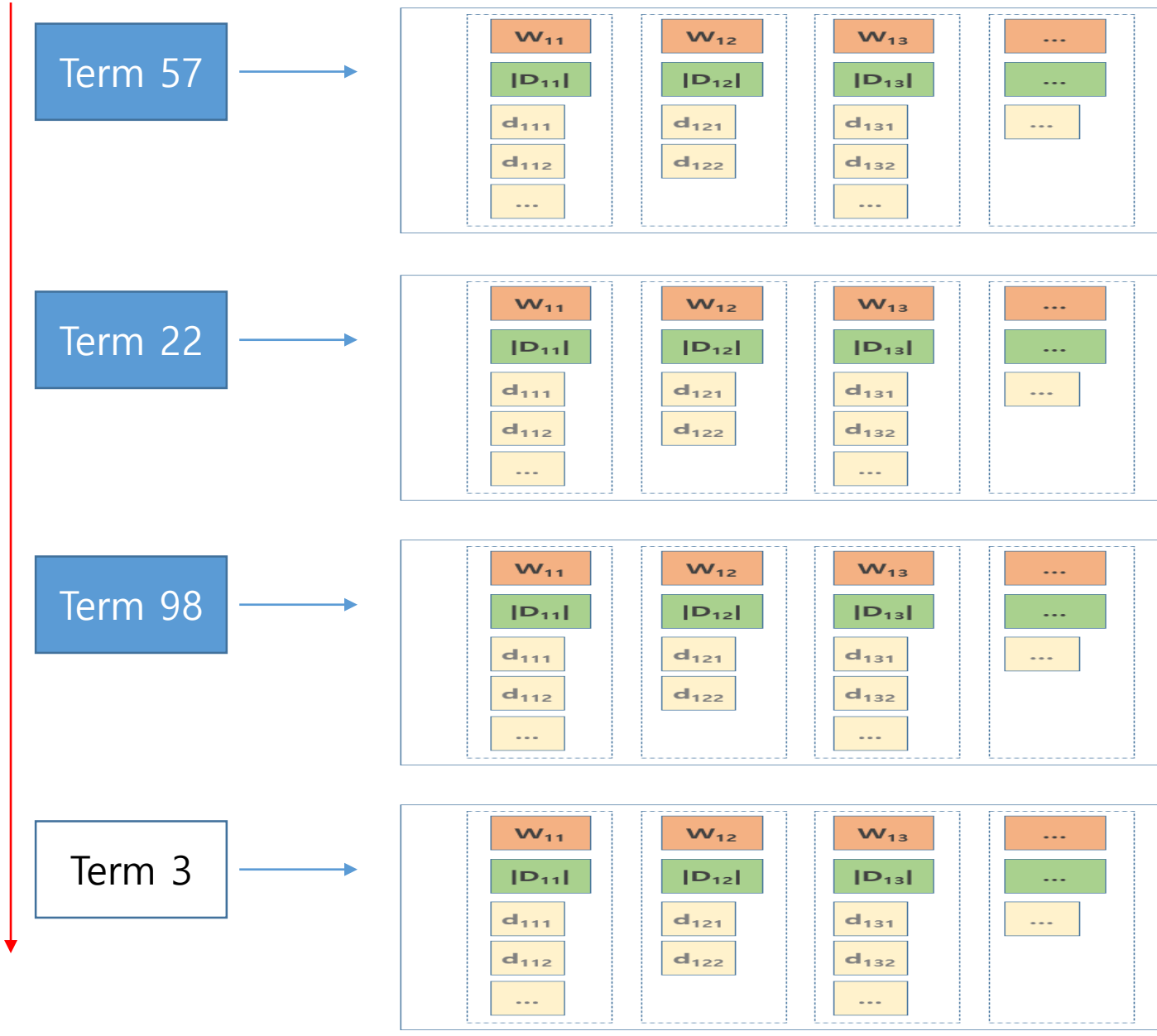
$$\cos(q,d) = \sum_{t \in T_1} w(q,t) * w(d,t) + \sum_{t` \in T_2} w(q,t`) * w(d,t`)$$

- **언제 term loop을 멈출** 것인가?

```
def find_similar_docs(query):
    scores = { }
    for t in query:
        for d in get_doc_having(term):
            scores[doc] += w(d, t) * w(q, t)
        if earlystop satisfied:
            break
    similars = sort_by_value(scores)
    Return similars
```

Query terms는 TF-IDF(q,t)에 의하여 내림차순으로 정렬이 되어있고,

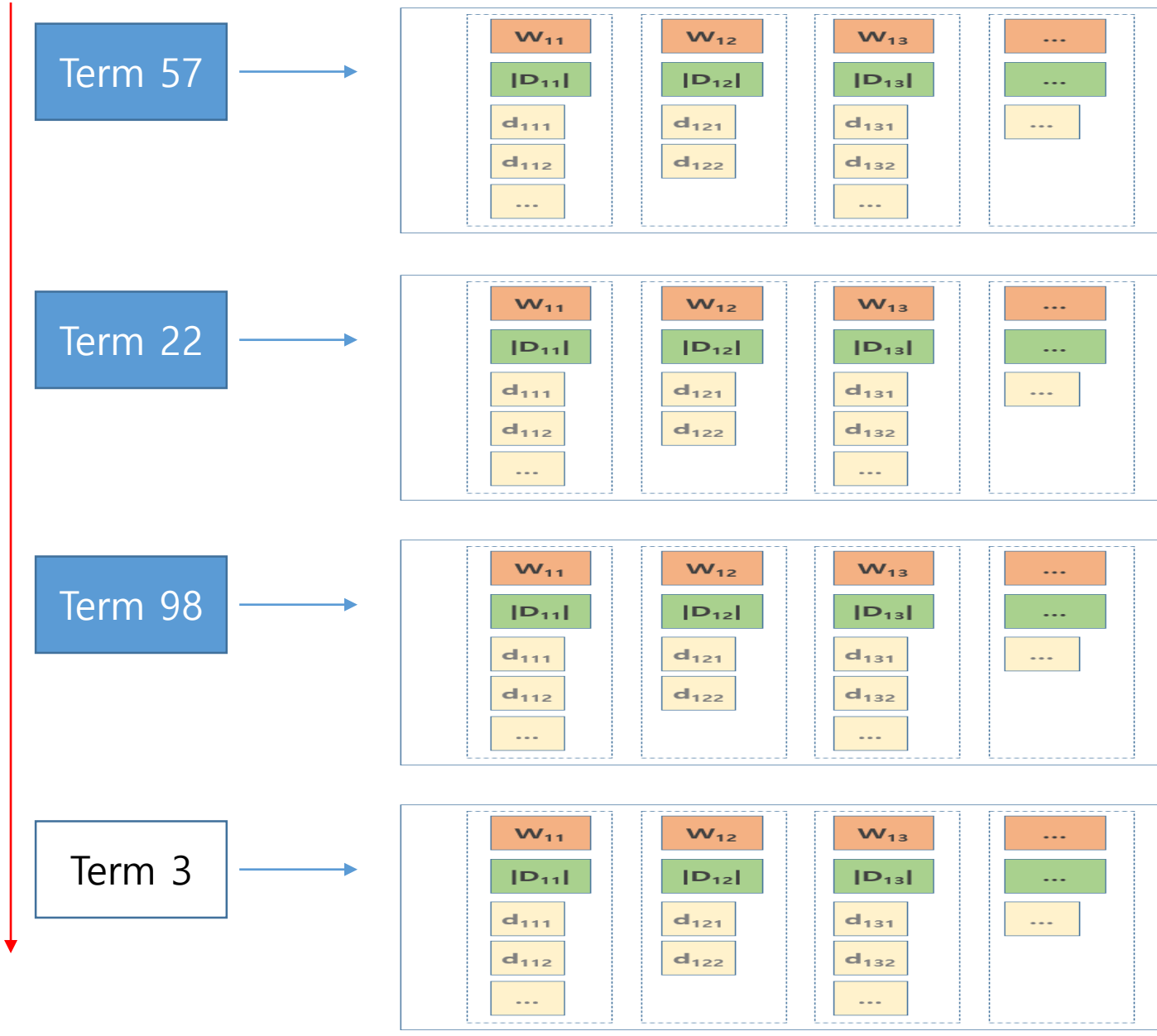
현재 term 98까지 이용하여 score[doc]을 계산. **Term 3은 고려해야하는가?**



지나치게 흔한 term t는 모든 문서의 scores를 높여줄 것이며,

어느 정도 score 계산이 끝났다면 (= remains가 어느 정도 작다면) early stop

```
def find_similar_docs(query):
    scores = { }
    remains = 1.0
    for t in query:
        remains = remains – w(q,t)²
        for d in get_doc_having(term):
            scores[doc] += w(d, t) * w(q, t)
        if remains * IDF(t) < threshold:
            break
    similars = sort_by_value(scores)
    Return similars
```



Performance

- Very sparse term document matrix
  - Shape = (3,221,405 by 172,392)
  - Sparsity = 0.0089 %
  - # doc = 3,221, 405
  - # vocab = 172,392
  - 문장 당 평균 15.4개의 단어가 들어있는 short sentences

Performance

Precision 10 @ k	Precision
K = 10	0.483
K = 20	0.623
K = 30	0.706
K = 50	0.774
K = 100	0.835
Query times = 23.215 mil. Sec.	

Recall 10 @ k	Recall	Query times [mil. sec.]
K = 10	0.483	21.61
K = 20	0.485	35.34
K = 30	0.753	42.82
K = 50	0.820	53.68
K = 100	0.917	75.76

* **Precision 10 @ k:**
  top 10개를 retrieval 했을 때,
  k 등수 안에 드는 문서의 개수

* **Recall 10 @ k:**
  k개의 문서를 retrieve 했을 때,
  True most similar 10 문서 중 포함 된 개수

Performance

- Recall 10 @ k 기준으로, most similar 10개의 문서를 찾기 위해서는 k=100으로 검색을 한 뒤, 100개의 문서에 대하여 true neighbors를 계산하여도 91.7% 찾을 수 있음

- FastCosine은 더 많은 이웃을 검색하거나 (= k를 크게 잡거나), filtering, early stop parameter를 조절하여 더 많은 계산을 수행할수록 더 좋은 검색 결과를 보장함

Recall 10 @ k	Recall	Query times [mil. sec.]
K = 100	0.917	75.76

Performance (LSH)

FastCosine

Precision 10 @ k	Precision
K = 10	**0.483**
K = 20	0.623
K = 30	0.706
K = 50	0.774
K = 100	0.835
Query times = 23.22 mil. Sec.	

* **Precision 10 @ k:**
top 10개를 retrieval 했을 때,
k 등수 안에 드는 문서의 개수

LSH
(4 tables, min bucket size = 10)

Precision 10 @ 10	**0.166**
Precision 10 @ 20	0.203
Precision 10 @ 30	0.217
Precision 10 @ 50	0.228
Precision 10 @ 100	0.235
Query times = **236.13 mil. Sec.**	

LSH
(4 tables, min bucket size = 30)

Precision 10 @ 10	0.161
Precision 10 @ 20	0.196
Precision 10 @ 30	0.214
Precision 10 @ 50	0.236
Precision 10 @ 100	0.269
Query times = 270.22 mil. Sec.	

LSH
(8 tables, min bucket size = 10)

Precision 10 @ 10	0.173
Precision 10 @ 20	0.201
Precision 10 @ 30	0.214
Precision 10 @ 50	0.230
Precision 10 @ 100	0.249
Query times = 419.20 mil. Sec.	

LSH
(8 tables, min bucket size = 30)

Precision 10 @ 10	**0.221**
Precision 10 @ 20	0.267
Precision 10 @ 30	0.294
Precision 10 @ 50	0.318
Precision 10 @ 100	**0.348**
Query times = **517.07 mil. Sec.**	

Performance

- FastCosine은 precision 10 @ 10 = 0.483을 얻는데 23.22 mil sec가 걸렸지만,
  LSH는 precision 10 @ 100 = 0.348을 얻는데도 517.07 mil sec를 씀

- LSH는 성능을 올리기 위하여 hash tables을 더 쌓는 수 밖에 없는데, 비용만 배수적으로 증가하며 성능 증가를 보장하지 못함