

코끼리 냉장고에 집어넣기: 실시간 추천엔진을 노트북에서 돌게 만들어보자

하용호

NUMBERWORKS 대표

NUMBER
WORKS



지금은 '넘버웍스'라는
data science firm을 만들었어요

어떤 순서로 말하나요?

1. 추천엔진의 틀에 대해 알아보자
2. 구글 논문대로 잘만드는 방법
3. 그거보다 더 멋지게 만들어보자.
4. 정리

추천엔진들의 틀에 대해 배우자.

Recommendation systems
come in

two types

Content-based

vs

Collaborative
Filtering

Content-based 는 어렵습니다.

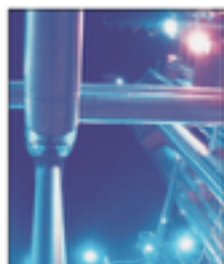
- 사전에 현업지식도 꽤 있어야 하고
- 개별 콘텐츠의 메타 데이터도 거의 없고
- 상품에서 그것을 뽑아내기도 어렵습니다.

그래서 모던한 추천엔진들은

Collaborative Filtering

을 씁니다.

Industry Report



Amazon.com Recommendations

Item-to-Item Collaborative Filtering

Greg Linden, Brent Smith, and Jeremy York • Amazon.com

Recommendation algorithms are best known for their use on e-commerce Web sites,¹ where they use input about a customer's interests to generate a list of recommended items. Many applications use only the items that customers purchase and explicitly rate to rep-

There are three common approaches to solving the recommendation problem: traditional collaborative filtering, cluster models, and search-based methods. Here, we compare these methods with our algorithm, which we call *item-to-item collaborative filtering*. Unlike traditional collaborative

Collaborative Filtering

역시 2가지로 나뉘게 됩니다.

Memory-based Model-based

보통은 둘다 조합(hybrid)해서 씁니다.

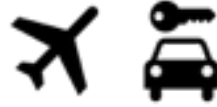
오늘은

Memory-based을

어떻게 쿡알만하게
만들 수 있는지 알아보시다.

memory-based 를 잘만드는 방법

YOU





YOU



recommend



Someone
LIKE YOU!



누가 가장 닮았냐 Jaccard similarity

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

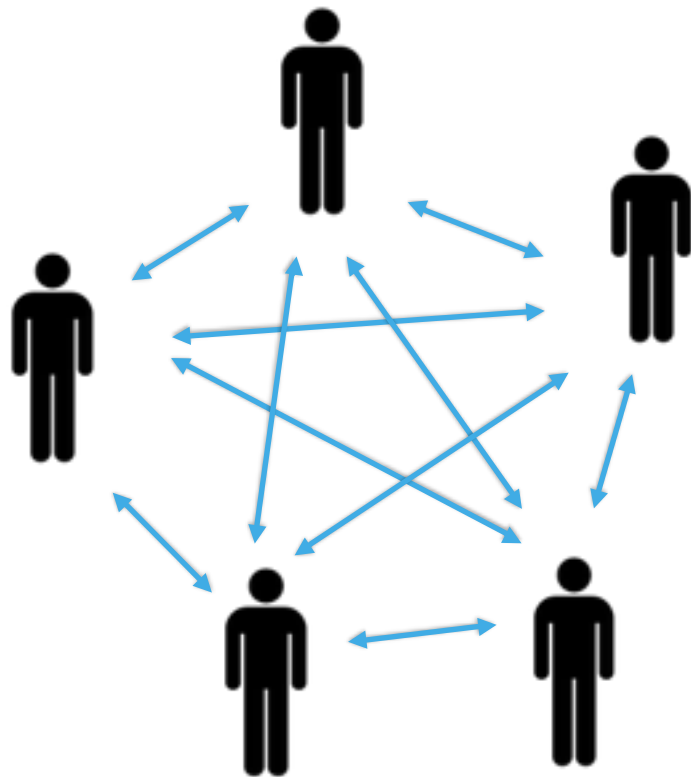
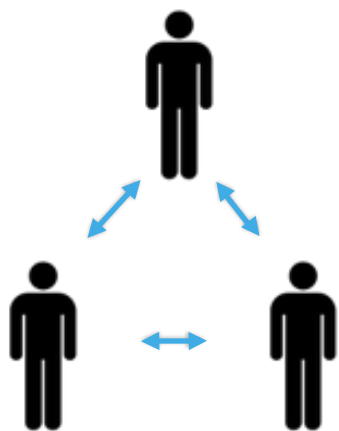
$A = [1,2,3,4]$ A가 좋아하는 상품

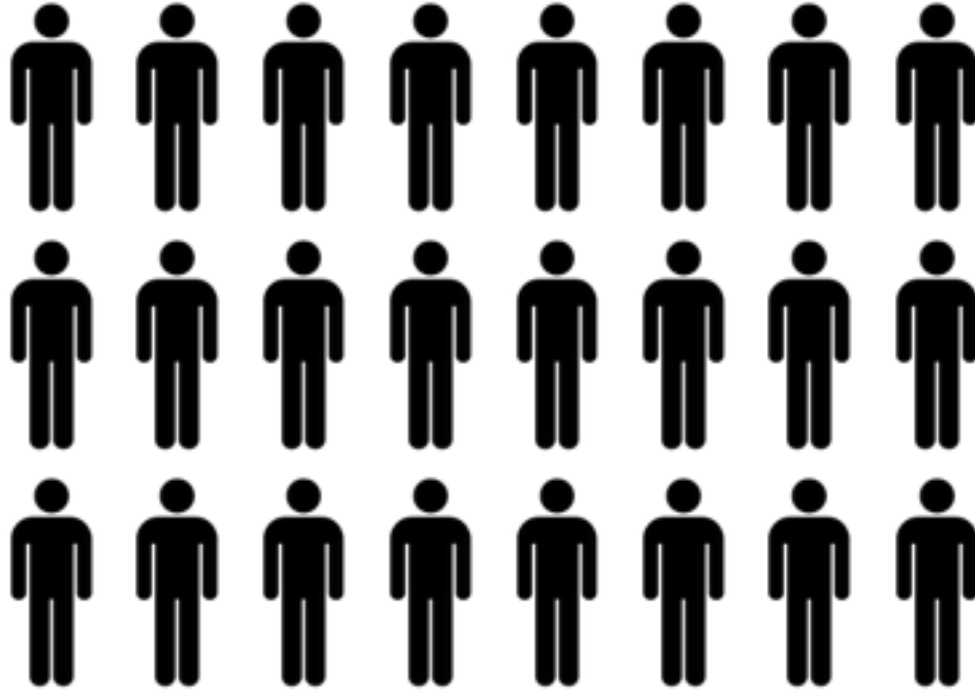
$B = [1,2,7]$ B가 좋아하는 상품

$$J(A,B) = \frac{|[1,2]|}{|[1,2,3,4,7]|} = \frac{2}{5} = 0.4$$

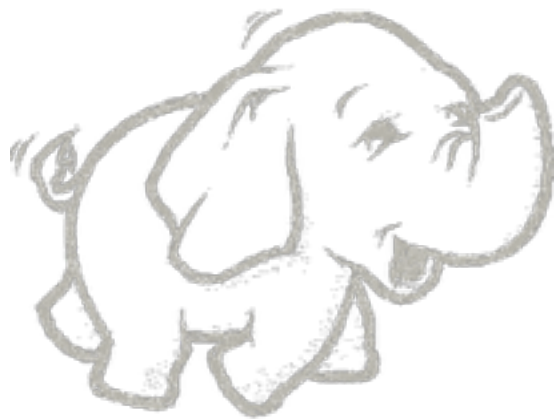
전체중에 겹치는 비율을 닮음으로 측정

but
size
matters





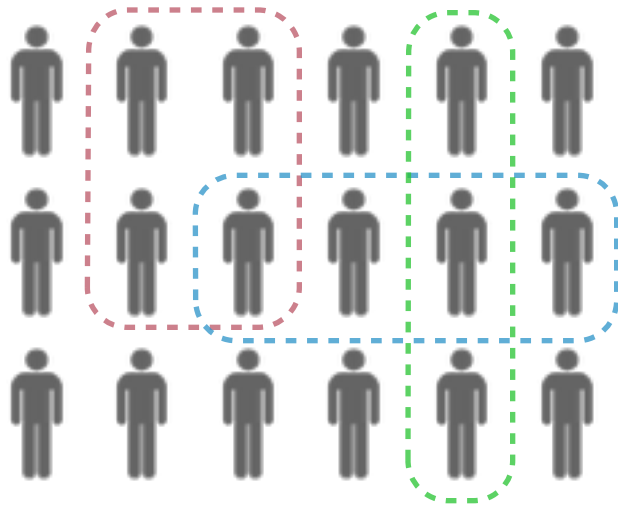
TOO MANY COMBINATIONS



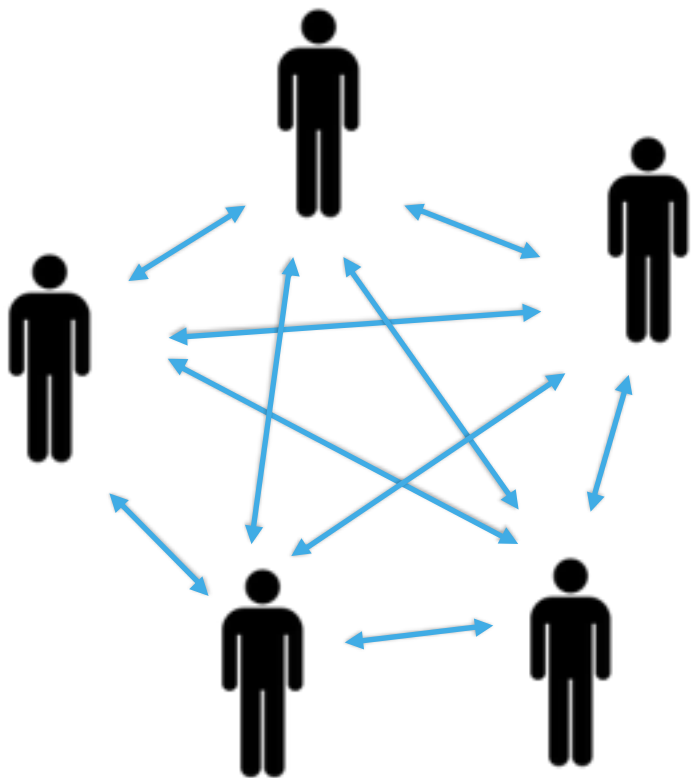
물론 이런데 쓰라고
Hadoop이 있기는 합니다만



이건 Hadoop에게도
너무 큰 계산입니다



비교 대상을 줄이기 위해
pre-clustering이 필요합니다



그런데 naive 한
clustering 방법들은
 $O(N^2)$

계산량이 많아
클러스터링 하려는데
클러스터링 계산량이 많다?

더 가벼운 클러스터링
방법이 필요!

pre-clustering에 관련된
마스터 피이쓰 논문이 있습니다

Google News Personalization: Scalable Online Collaborative Filtering

Abhinandan Das
Google Inc.
1600 Amphitheatre Pkwy,
Mountain View, CA 94043
abhinandan@google.com

Mayur Datar
Google Inc.
1600 Amphitheatre Pkwy,
Mountain View, CA 94043
mayur@google.com

Ashutosh Garg
Google Inc.
1600 Amphitheatre Pkwy,
Mountain View, CA 94043
ashutosh@google.com

Shyam Rajaram
University of Illinois at Urbana
Champaign
Urbana, IL 61801
rajaram1@ifp.uiuc.edu

ABSTRACT

Several approaches to collaborative filtering have been studied but seldom have studies been reported for large (several million users and items) and dynamic (the underlying item set is continually changing) settings. In this paper we describe our approach to collaborative filtering for generating personalized recommendations for users of Google News. We generate recommendations using three approaches: collaborative filtering using MinHash clustering, Probabilistic Latent Semantic Indexing (PLSI), and covisitation counts. We combine recommendations from different algorithms using a linear model. Our approach is content agnostic and consequently domain independent, making it easily adaptable for other applications and languages with minimal effort. This paper will describe our algorithms and system setup in detail, and report results of running the recommendations engine on Google News.

me something interesting. In such cases, we would like to present recommendations to a user based on her interests as demonstrated by her past activity on the relevant site.

Collaborative filtering is a technology that aims to learn user preferences and make recommendations based on user and community data. It is a complementary technology to content-based filtering (e.g. keyword-based searching). Probably the most well known use of collaborative filtering has been by Amazon.com where a user's past shopping history is used to make recommendations for new products. Various approaches to collaborative filtering have been proposed in the past in research community (See section 3 for details). Our aim was to build a scalable online recommendation engine that could be used for making personalized recommendations on a large web property like Google News. Quality of recommendations notwithstanding, the following requirements set us apart from most (if not all) of the known

이 논문의 핵심 아이디어는

Use MinHASH as LSH

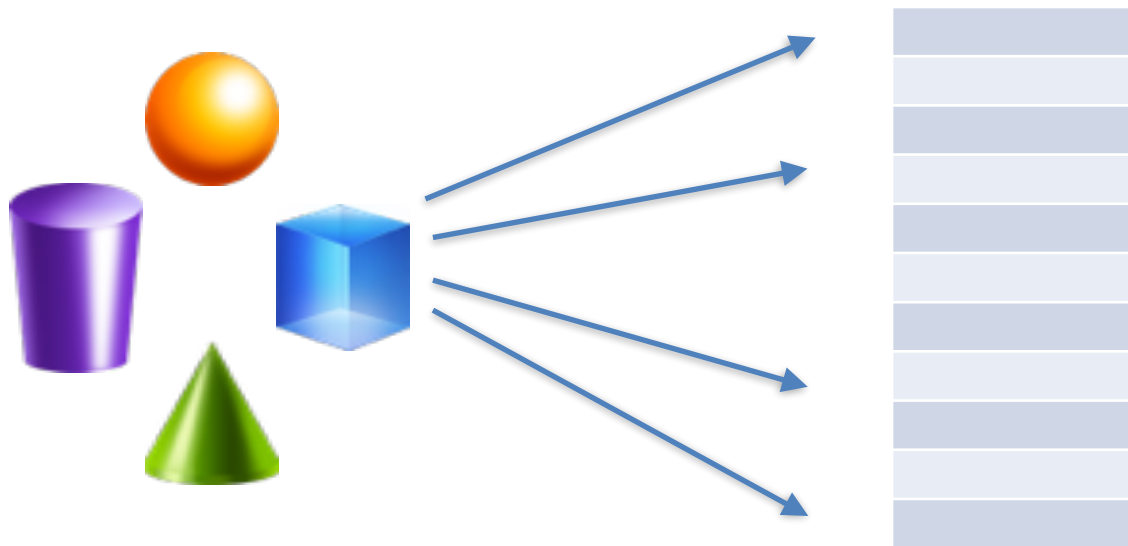
(min-wise independent permutations locality sensitive hashing)

이건 또 뭘소리야?

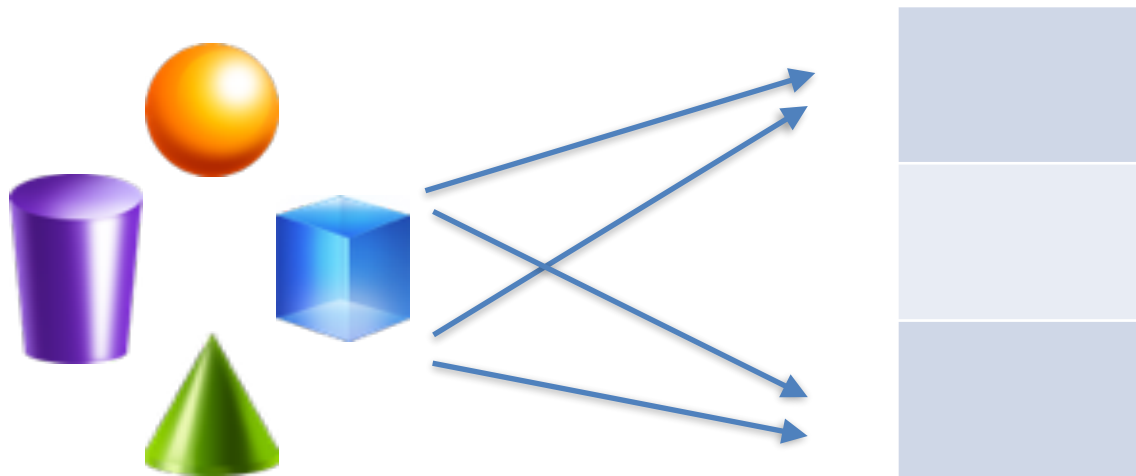
알아보자.

Locality Sensitive Hashing

기존 자료구조에서 hash는
충돌나지 않게 넓게 뿌리는 것이었지만



만약 적은 버킷에 일부러
비슷한 것들끼리 충돌나게 뿌리면?



clustering처럼 동작하게 된다.

(locality sensitive hashing)

HOW DOES LSH WORK?

Make hash functions

Hash function #1



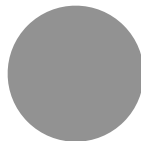
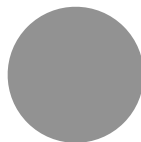
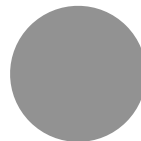
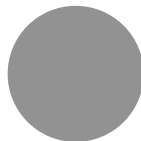
Hash function #2

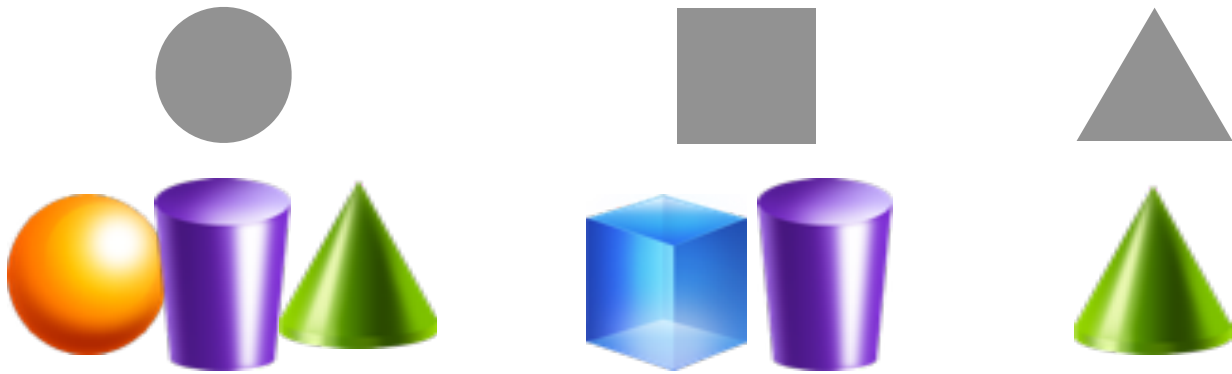


Hash
Func #1



Hash
Func #2





클러스터링은 원래 $O(n^2)$ 이지만

hash snapshot으로 $O(n)$ 짜리

클러스터링

TIME



1) 계산할 녀석과
비슷한 애들이 있는
클러스터를 찾는다

2) 상세한 비교는 $O(n^2)$
하지만 부담되지 않는
 n 에서 동작한다.

SIZE



원래 hash function은
원본이 조금만 달라도 훌훌 달라지는게 좋은 hash였다.

그런데 이제는 기존과 반대로
원본이 닮았으면, 결과값들도 닮아야 하는
특수한 hash function이 필요하다.

답음의 종류가 많다.

- cosine similarity
- hamming distance
- euclidean distance
- jaccard similarity

MinHash는
Jaccard similarity를
유지하는 타입의
LSH

이런 놈이 필요했다!

	A	B	C
r1	1	0	1
r2	0	1	1
r3	0	0	1
r4	1	1	0
r5	1	0	0
r6	0	1	1

random permutation

hash1	hash2	hash3
r3	r6	r5
r5	r1	r4
r1	r3	r6
r2	r2	r2
r4	r4	r1
r6	r5	r3

	A	B	C
h1	2	4	1
h2	2	1	1
h3	1	2	3

random하게 permutation한 순서로 읽어가며 첫 1의 index를 구한다.

jaccard = intersect/union

jaccard(A,C) = $1/6 = 0.16$

jaccard(B,C) = $2/5 = 0.4$

sim = intersect/length

sim(A,C) = $0/3 = 0$

sim(B,C) = $1/3 = 0.3$

실제로는 random permutation을 만드는 것도
계산량이 많이 필요하게 되므로
universal hash를 잔뜩 random generation
이를 random permutation의 대리자(proxy)로
사용하곤 한다.

뭔가 말이 어려워 보이지만
하여간 저렴한 계산으로
minhash를 얻을 수 있습니다.

이건 일종의 dimension reduction

	<i>A</i>	<i>B</i>	<i>C</i>
<i>r1</i>	1	0	1
<i>r2</i>	0	1	1
<i>r3</i>	0	0	1
<i>r4</i>	1	1	0
<i>r5</i>	1	0	0
<i>r6</i>	0	1	1

6차원
↓

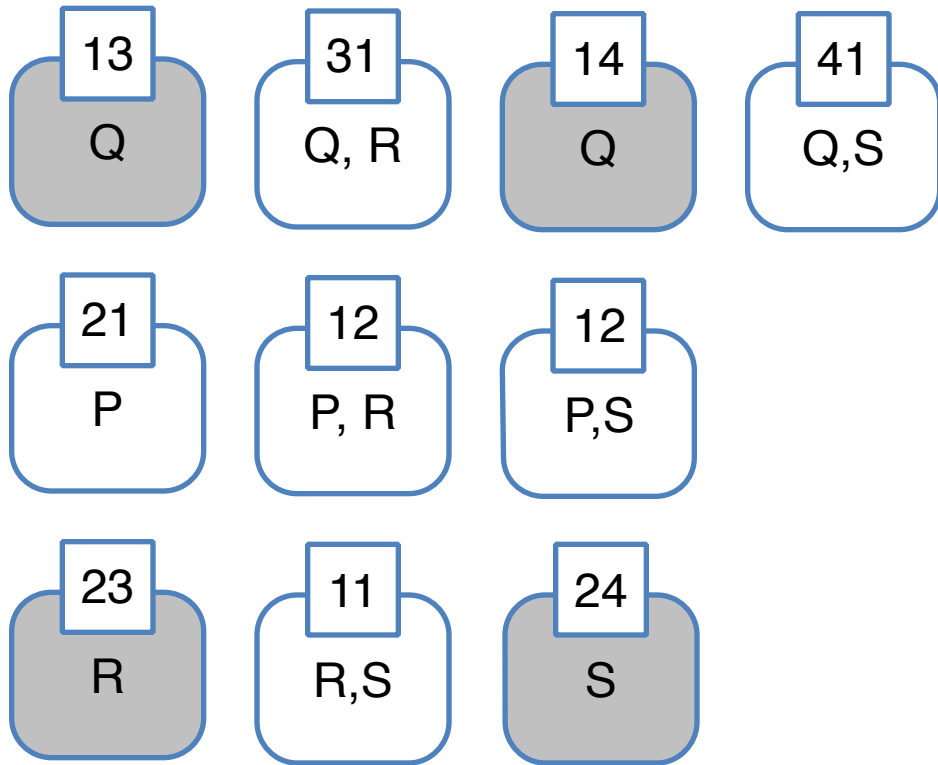
3차원
↓

	<i>A</i>	<i>B</i>	<i>C</i>
<i>h1</i>	2	2	1
<i>h2</i>	2	1	1
<i>h3</i>	1	2	3

hash되어 나온 값을 signature라 하고
이것들을 적당히 concatenate시켜서 cluster id 삼는다.

	Q	P	R	S
s1	1	2	2	1
s2	3	1	3	1
s3	1	2	1	2
s4	4	1	1	4

‘s1-s2’, ‘s2-s3’,
‘s3-s4’, ‘s4-s1’



그래서 typical한
Batch Implementation은
다음과 같습니다.

(item to item 케이스로 설명)

minhash로 클러스터 만들기

for 후보 in 모든 후보들:

- 그 후보의 모든 click stream 로딩하기

- 로딩한 것으로 minhash signature 생성하기

- signature들을 concatenate해서 다량의 cluster id

- 이 후보를 해당 cluster id들에 집어넣기

for cluster in 만들어진 클러스터들:

- if length(클러스터) > threshold:

- 클러스터의 멤버십 정보 저장

클러스터 정보로 추천하기

타겟 item이 존재함

그 타겟 item의 minhash값을 가져옴

minhash값을 concatenate해서 해당되는 cluster 들 다 찾음

for cluster in 타겟이 포함된 클러스터들:

 클러스터에 포함된 다른 뽑은 item들 로딩

 그 item들의 click stream을 로딩

 click stream정보를 기초로 모든 pair 유사도 계산

각 클러스터로 부터 모여온 것중 유사도 top N들을 선출

It is typical
implementation.
but not attractive :(

더 멋지게
만들어 보자!

아까 구현은 2개의 문제가 있다.

느려

구려

Heavy I/O 가 잦다 : 빨간색이 I/O

minhash로 클러스터링

for 후보 in 모든 후보들:

그 후보의 모든 click stream 로딩하기

로딩한 것으로 minhash signature 생성하기

signature들을 concat해서 다량의 cluster id

이 후보를 해당 cluster id들에 집어넣기

for cluster in 만들어진 클러스터들:

if length(클러스터) > threshold:

클러스터의 멤버십 정보 저장

클러스터 정보로 추천하기

타겟 item이 존재함

그 타겟 item의 minhash값을 가져옴

minhash값을 concat해서 cluster id 리스트작성

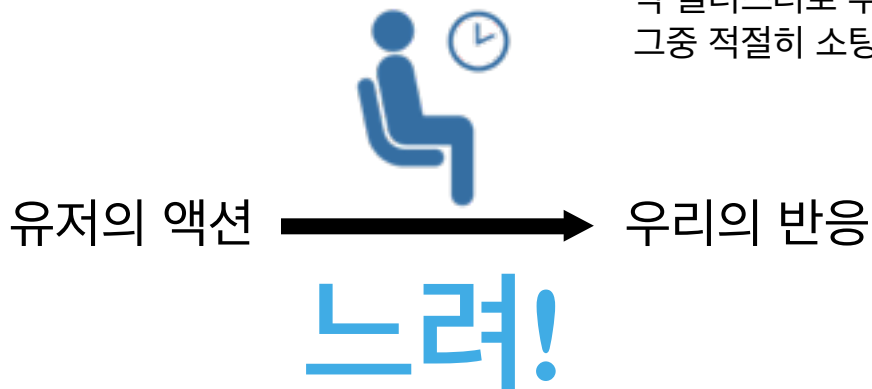
위를 통해 타겟이 포함된 cluster들을 다 찾음

for cluster in 타겟이 포함된 클러스터들:

클러스터에 포함된 다른 item들 로딩

그 item들의 click stream을 로딩

click stream정보를 기초로 모든 pair 유사도 계산
각 클러스터로 부터 모여온 것중 유사도 top N들을 선출
그중 적절히 소팅하여 추천



클러스터를 쓴다

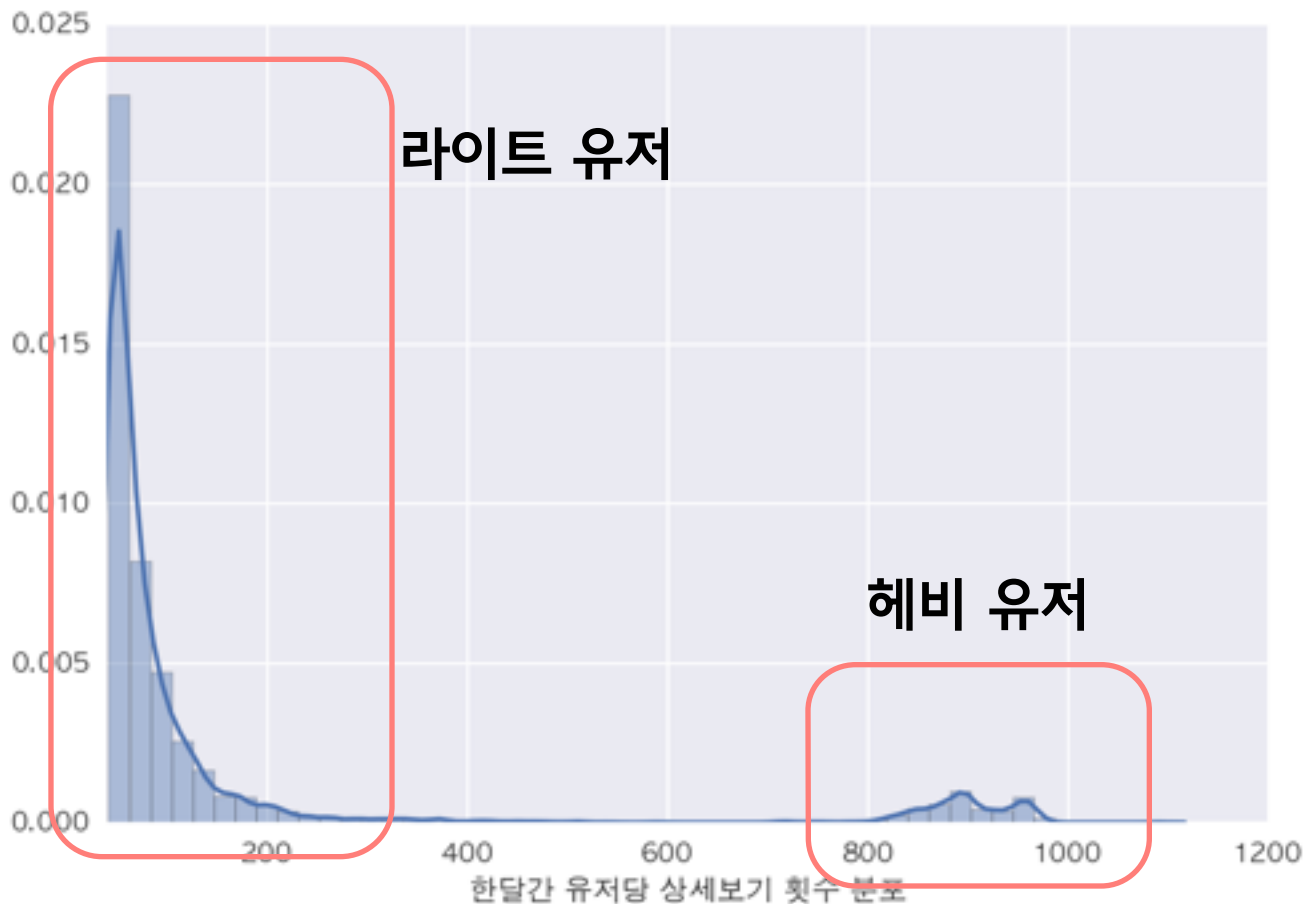
장점 : 후보를 한정한다. - speed gain

단점 : 후보를 한정한다. - quality loss

어떻게 하면
엄청나게 빠른 속도로
퀄리티 손상없이
추천을 만들어낼 수 있을까?

왜 I/O 문제가 되는 것일까?
상품과 유저의 부익부 빈익빈 현상때문이다.

대부분의 서비스는 이런 양상 : user당 view수



대부분의 서비스는 이런 양상 : item당 view수

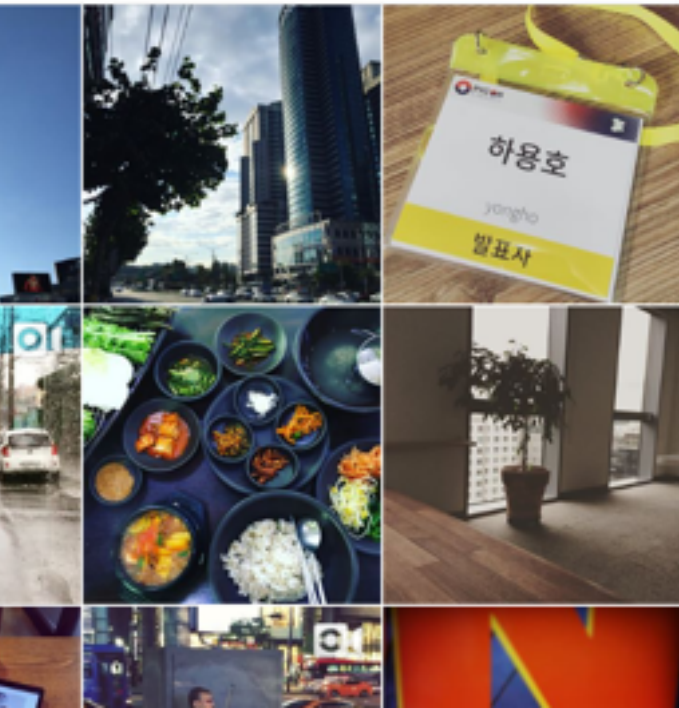


100
게시물

160
팔로워

183
팔로잉

보통은 이정도



SKT

오후 7:10



DLWLRMA



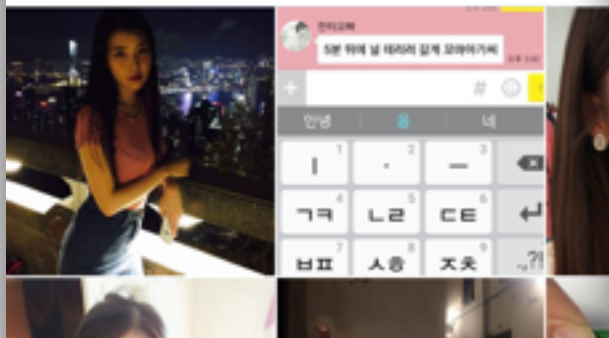
56
게시물

1.3m
팔로워

✓ 팔로잉

이지금

아이유님



SKT

오후 7:10



JUSTINBIEBER



2448
게시물

38.5m
팔로워

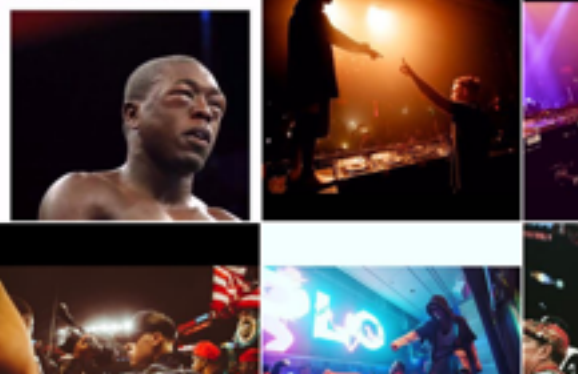
저스틴비버

Justin Bieber

Help change the world. New Single #What
smarturl.it/iWDYM



Waking up during the night and your
phone brightness be at 100%



매우 인기있는 유저나 아이템 존재

= 비교대상으로 자주 등장 (인기있다니까)

= 엄청나게 긴 click stream

= 로딩하는데 시간 걸린다

= 공간모자라 캐시 로딩한 것도 날라간다.

= 각종 page out도 발생한다.

=퍼포먼스가 떨어진다.

클릭 스트림의 길이

인기 아이템



보통 아이템



보통 아이템



보통 아이템



이 클릭 스트림의 길이를
어떻게든 해야
등장할 때마다
퍼포먼스 갱판 치는 것을
막을 수 있다.

해결 방법

= 클릭 스트림 원본 말고, 짧은 대체본

= use good dimension reducer

= 대체본끼리 비교해 바로 유사도를 알아야.

= 어 그런거 이미 알고 있잖아?

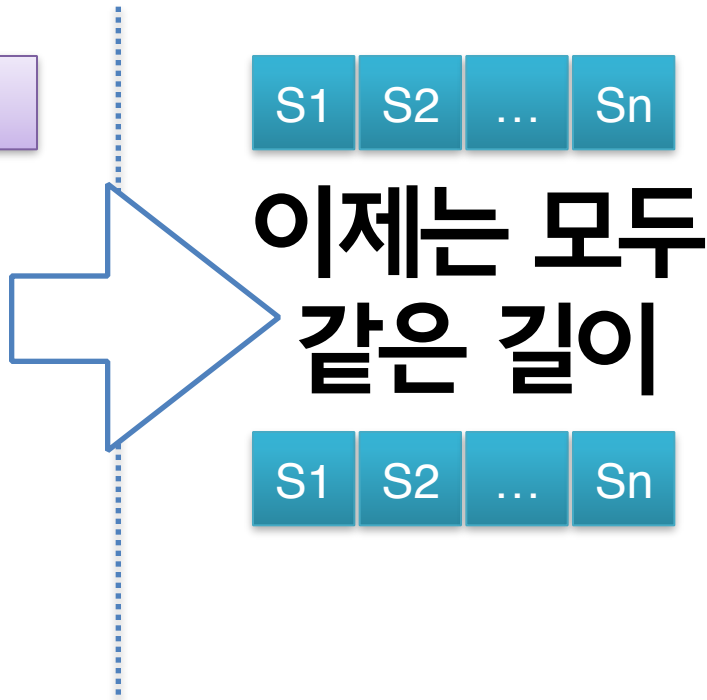
= minhash

hash function 수 n 만큼 signature 생성

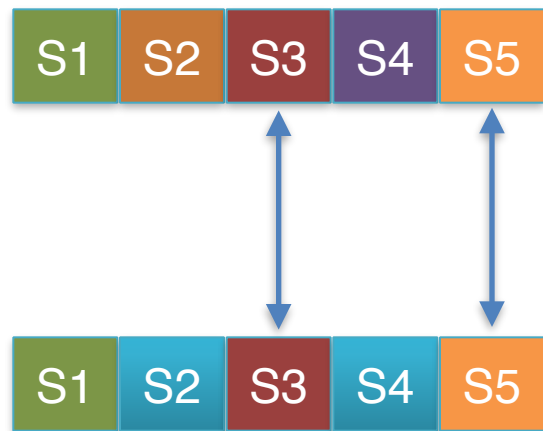
인기 아이템



보통 아이템



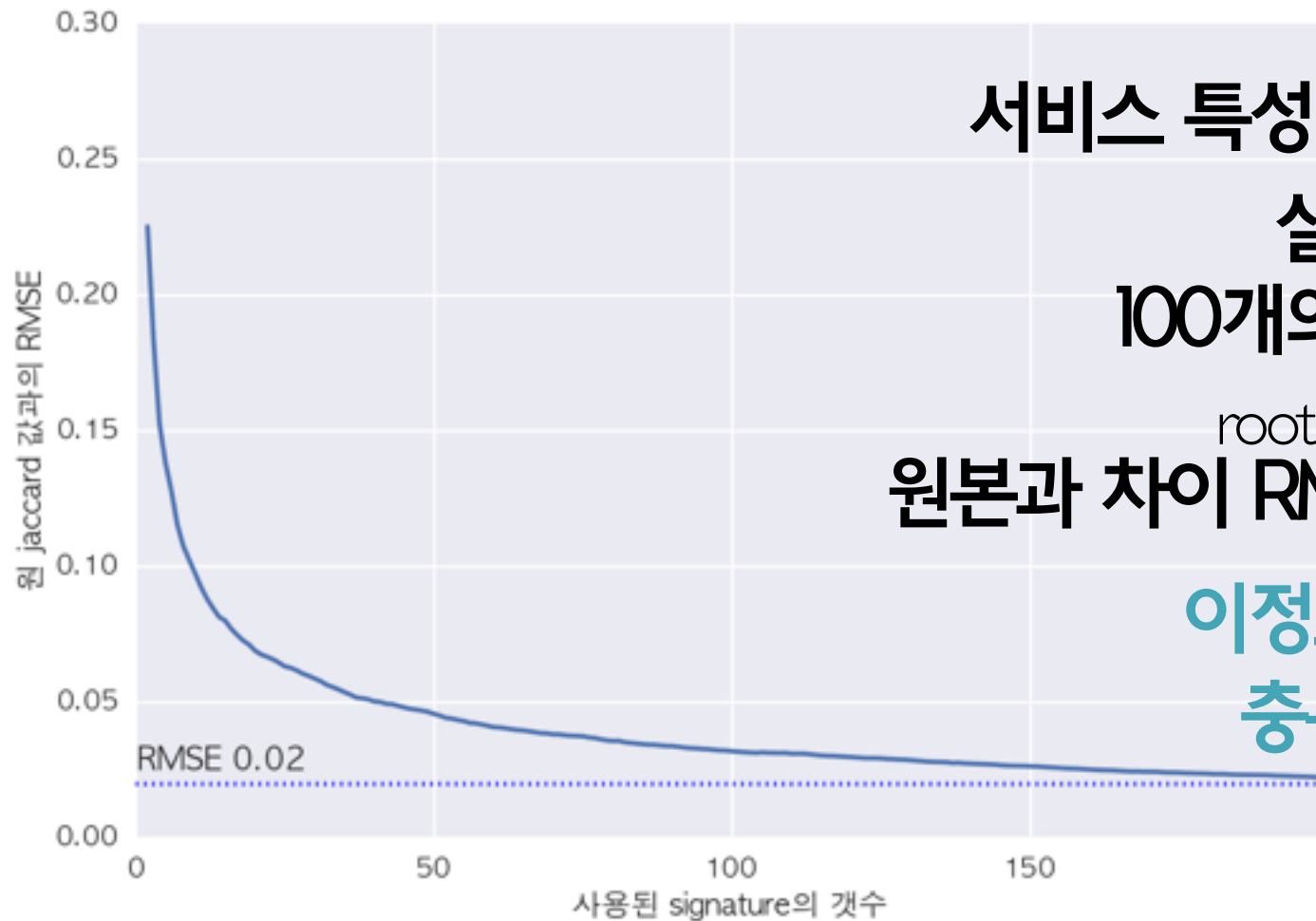
signature들 사이 겹치는 sig의 비율이 곧 jaccard



추정된 jaccard
coefficient
 $\frac{2}{5}$
 $= 0.4$

그런데 아까도 봤지만, 이 추정값과
실제 jaccard량은 오차가 있던데?

더 많은 hash func을 써서
signature의 길이를 길게 할수록
오차는 줄어든다!



서비스 특성마다 다르지만
실험데이터에서
100개의 signature면
root mean square error
원본과 차이 RMSE 0.03 유지

이정도면 쓰바라시
충분히 근사한다!

구글 논문과 반대의 어프로치

구글 논문

signature를 소량 생성
적절히 충돌시켜
미리 clustering 하자
실제 계산은
click stream을 사용함

새로운 방법

signature를 대량 생성
충돌을 피하고
실제 계산은
click stream 대신
signature overlap을 사용함

이 방법의 장점

다량의 건수를 in-memory에 fit가능
자료구조상 실시간 반영 추천이 가능

+다량의 건수를 in-memory에 fit가능
앵간하면 전부 disk io없이 계산

+메모리 얼마가 필요한지 예측 가능
예전에는 인기아이템의 클릭스트림이
얼마나 용량을 차지 할지 몰랐다.

item한개가 4byte짜리 시그니처 100개
개당 부가 공간 200 byte 더 있다 치자

$$100 \times 4 + 200 = 600 \text{ byte}$$

$$\text{메모리 1G가 있으면 } 1024^3 / 600 = 1,789,569$$

1기가당 180만 종류 item의 click stream을
메모리에 들고 있을 수 있다! (그런 규모 많이 없다)

+우리를 느리게 만드는 건 대량 배치 Job
“이거 다 돌아야 반영되어요”

+빛은 쌓지 말고 짹짹 분할상환하는게 좋다.
“새로운 클릭이 쌓이면
그거 포함해서 다시 전부 돌려야
하지 않나요?”, “방법이 있습니다”

새로운 click이 일어나면
기존에 계산해 놓은 signature버리고
다 새로 계산 해야 하나?

No!

minhash는 'min 함수'의 'chain'이다.
때문에 좋은 성질을 가지고 있다.

$$\min(A,B,C) = \min(\min(A,B),C)$$

Associative property

결합법칙

$$\text{MinHash}(H_1, H_2, H_3, \dots, H_n) = \text{MinHash}(\text{MinHash}(H_1, \dots, H_{n-1}), H_n)$$

새 데이터가 들어오면 누적 적용하면 된다

Idempotence

멱등법칙

$$\text{MinHash}(\text{MinHash}(H_1, H_2), H_3) = \text{MinHash}(H_1, H_2, H_3)$$

$$\text{MinHash}(\text{MinHash}(H_1, H_2, H_3), H_2) = \text{MinHash}(H_1, H_2, H_3)$$

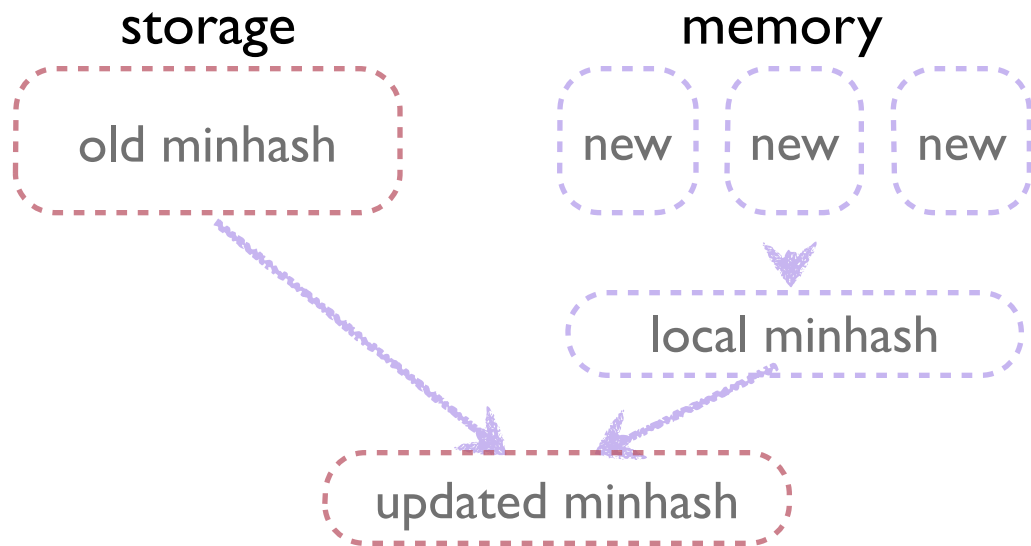
에러나도 안전하니 다시 적용하면 된다.

처음부터 지금까지 배치로 돌린것
= 맘 편하게 계속 누적시켜 적용한 것

click이 들어올때마다 지금 적용하자!

전체 배치는 부담되지만, 조각들은 가볍고 빠르다!

이 성질을 이용하면 high TPS도 대응



급격히 들어오면
모아서 그것만으로
선처리한다.
micro batch

뒤에 합쳐도
순차적으로 처리한
것과 같다.
(결합법칙)

save minhash(110)[1 DB write] =
load minhash(100)[1 DB read] x
compute minhash(101~110)[input buffer]

자~ 자료는
아주 작고 단단하게 잘 만들었다.
하지만 이것의 계산
어떻게 최적화 안될까?

자료는 잘 줄였다.

하지만 n^2 의 비교?

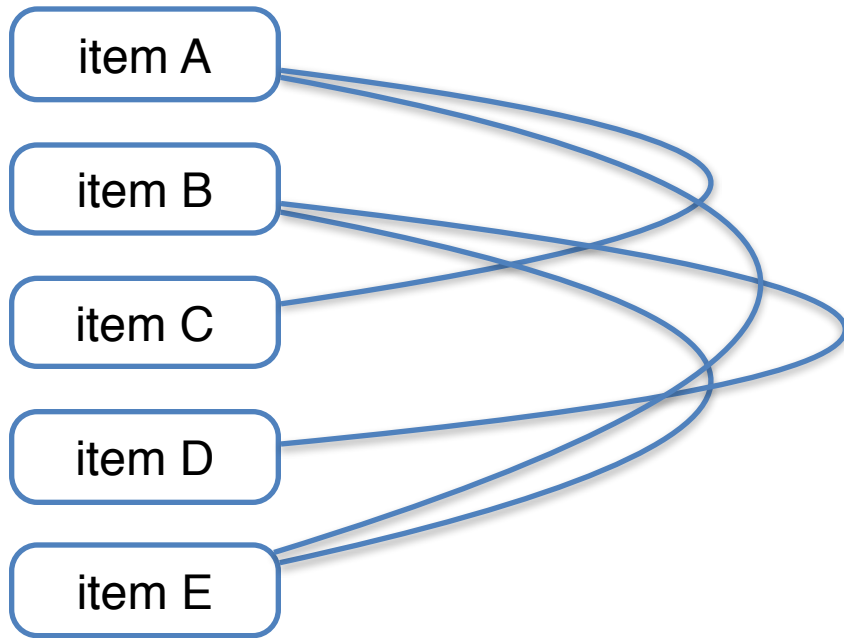
S1	S2	...	Sn
----	----	-----	----

S1	S2	...	Sn
----	----	-----	----

S1	S2	...	Sn
----	----	-----	----

S1	S2	...	Sn
----	----	-----	----

S1	S2	...	Sn
----	----	-----	----




이를 해결 하기 위해
Secondary Index를 만든다.

이렇게 signature로 표현 될 때


Secondary-Index KV Storage

Sig위치와 값 pair를 Key로 하고
그걸 가지는 item리스트를 Value

	Sig1	Sig2	Sig3	Sig4
item A	923	1032	58	74
item B	87	1032	123	80
item C	923	872	58	80



Key	Value
Sig1-923	A-C
Sig2-872	C
sig3-58	A-C
sig4-80	B-C



Value에 등장한 item이름 횟수만큼이
내 sig와 겹치는 갯수이고 = jaccard에 비례

$$\begin{array}{l} A : 2\text{회}/4\text{sigs} \\ B : 1\text{회}/4\text{sigs} \end{array} = \begin{array}{l} A : 0.5 \\ B : 0.25 \end{array}$$

Secondary Index lookup만으로
Jaccard를 계산할 수 있다.

오오오~!

requirement:

minhash재계산 업데이트마다

2nd idx도 매우 매우 잦은 업데이트

= 반드시 메모리에서 처리되어야 겠구나!

Large Secondary-Index는 어떤 저장장치가 적절할까?



REDIS의 많은 자료구조중에 뭘 써야 할까?

Key	Value
Sig1-923	A-C
Sig2-872	C
sig3-58	A-C
sig4-80	B-C

이러한 membership
정보를 저장해야 한다.

Redis Data structure candidates

Strings - plain K/V

VS

Sets - support add, remove, union, intersection



좋아보인다!

사실 멤버십 정보를 string으로 만들어 저장하는 건 귀찮다.

sig45 Tom Jerry Robert Jack

[write]

string으로 만들어 저장

json.dumps(data)

“[Tom, Jerry, Robert, Jack]”

[read]

가져온 것 다시 복호화

“[Tom, Jerry, Robert, Jack]”

json.loads(data_str)

하지만 승자는 String이다. 왜?!

이 경우에는 String이 Set보다 훨씬 좋다.

I) ^{string} Order! $O(1)$ vs ^{set} $O(N)$

```
In [24]: %time gs.load_benchmark('user','key')  
CPU times: user 0.32 s, sys: 0.03 s, total: 0.34 s  
Wall time: 0.42 s
```

```
In [25]: %time gs.load_benchmark('user','set')  
CPU times: user 32.34 s, sys: 0.13 s, total: 32.47 s  
Wall time: 33.88 s
```


‘redis string’ 은 mget 명령이 있습니다.
(multiple get at once)

2) N call round trip -> 1 call

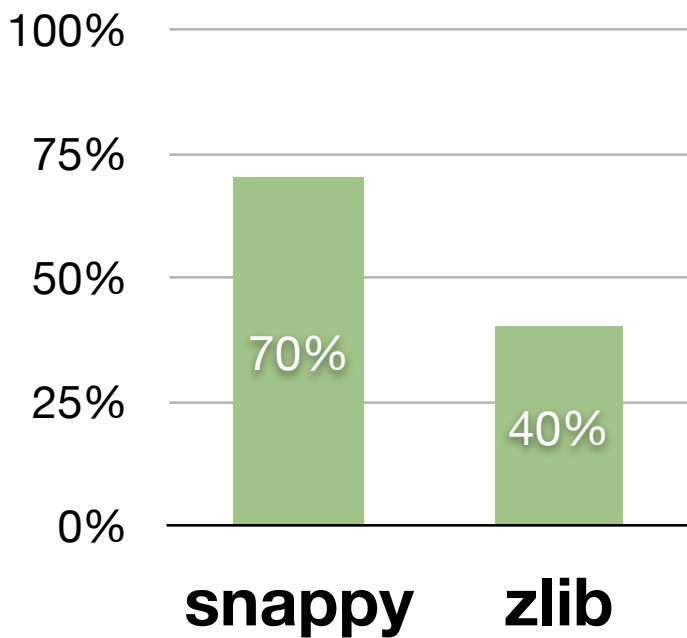
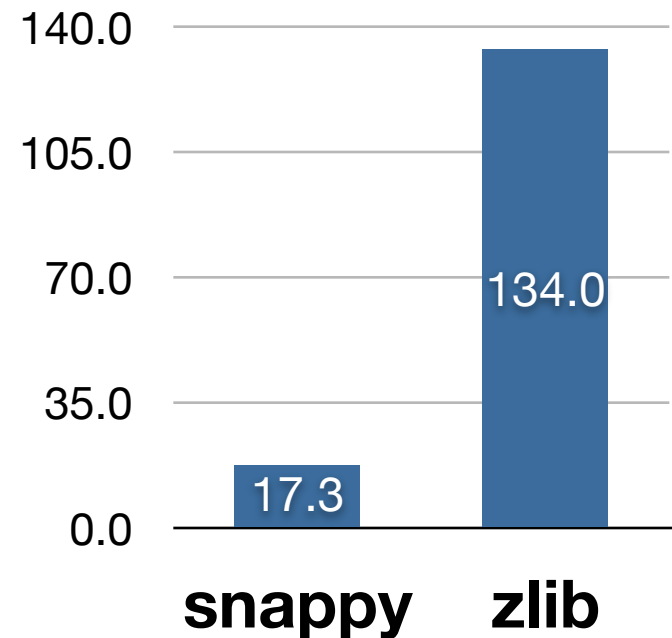
```
In [9]: %timeit [redis.get(s) for s in sigs]  
100 loops, best of 3: 9.99 ms per loop
```

```
In [10]: %timeit redis.mget(sigs)  
1000 loops, best of 3: 759 us per loop
```

3) string 은 압축을 적용하기에도 편합니다.
= 메모리에 더 많이 들고 있을 수 있단 듯이죠

compress speed(μ s)

size(%)



snappy쓰면
압축했는지 표도
안날정도로 빠름

4) redis 는 pipe기능을 이용해 transaction 이 됩니다.

요걸로 set의 기능들을 흉내 내는게 가능하죠

case) sig2-1032의 A가 빠지면서 sig2-973에 추가 - atomic하게

minhash signature 업데이트

	sig1	sig2	sig3	sig4
item A	923	1032 973	58	74
item B	87	1032	123	80
item C	923	973	58	80

Secondary-Index
key: value:
pos-value member

sig2-1032	A-B
sig2-973	C
sig2-1032	B
sig2-973	A-C

가질 수 있는 공금증

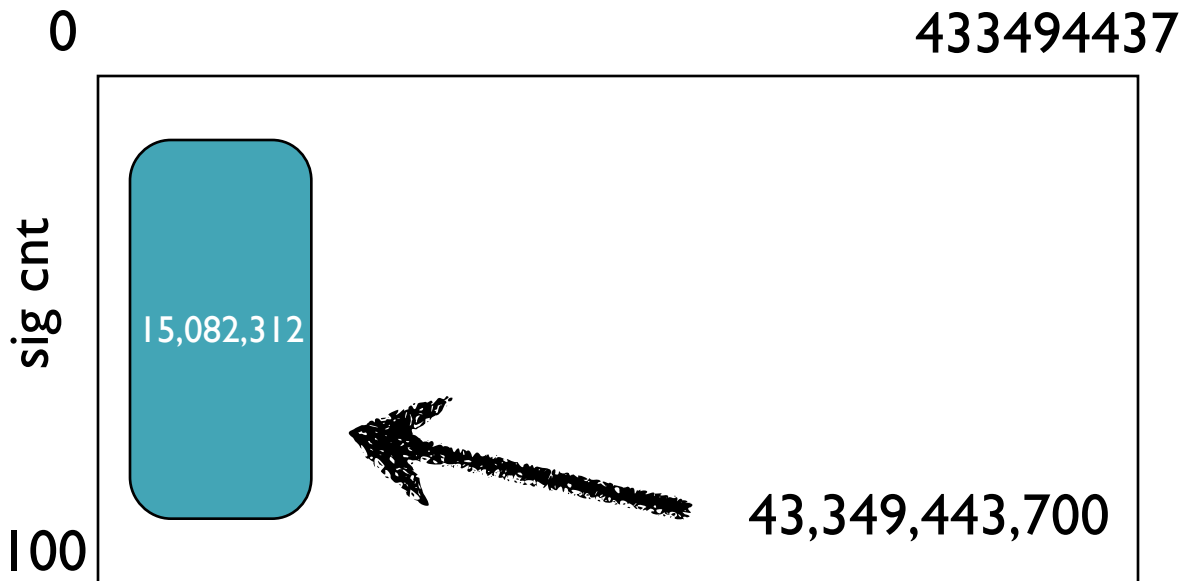
2nd index key space가 걱정입니다?

이론적으로는 $\text{hash max val} * \text{sig cnt}$ 의 key가 필요

hash max를 433,494,437인 Fibonacci primes을 쓰고
100개의 signature를 만드는 경우

이론적으론 433,494,437,00 개의 key영역 필요. 하지만.

minhash 는 min의 chain이기 때문에,
시간이 지나면 작은 영역에 수렴
0개의 member를 가진 key는 수시로 삭제



example case)

3,727,344 products,

100 sig =>

15082312 keys

압축적용 후, KV모두
메모리 2G 미만 차지

REDIS를 이용하면
transaction이 지원되면서
빠른 update가 되는
Secondary Index가
가능해집니다.

정리

item Z에 user click이 발생했을 때 반영법

minhash를 min하게 유지시킨다.

click 발생시 조금씩 바로 처리

그것의 signature 로딩

Z

183

1032

942

80

click 의 minhash계산

click

87

2043

123

300

새롭게 얻은 가장 낮은 값으로
signature 및 2nd index 변경

new

87

1032

123

80

Secondary Index 가 있을 때 추천 flow

어떤 item Z와 닮은 것을 찾아야 한다면?

그것의 signature 로딩

item Z

87	1032	123	80
----	------	-----	----

redis.mget으로 한방에 가져옴

갯수만 카운트하면, 찾기끝

Sig1 - 87	A-B-D-F-Z
Sig2 - 1032	C-G-Z
Sig3 - 123	A-B-D-Z
Sig4 - 80	B-D-F-Z

A:2/4 =0.5	B:3/4 =0.75
C:1/4 =0.25	D:3/4 =0.75

코끼리는 냉장고에 들어갔는가?

필요한 CPU Power

REDIS 1 core

minhash 유지 1 core

추천 계산 1 core

필요한 Mem Size

minhash용 1G

2nd index용 2G

기타 1G



요정도면 되겠습니다.

사실 진정한 추천엔진은
이런 단순한 Memory based 말고도
굉장히 많은 요소들과
(ALS, NMF, Markov Chain등등)
적절한 프로파일링들이 잘 섞여야 합니다.



그럼 이것은 낚시냐?

No!

99.9% 정확도로 다른 큰 이득을 취하는 것

logo.bmp

NUMBER
WORKS

1169KB

$\xrightarrow{1/22}$

logo.jpg

NUMBER
WORKS

51KB

크고 어려운 일은, 작은일로 평소에 하는 것

거대한
인덱스 빌딩

분할 상환
→
Amortized

평소에
자잘한 업데이트



강의 시작 전에는
대형 클러스터가 필요했지만
강의가 끝날 무렵
필요 없어졌습니다.

오늘의 교훈

확률적 자료구조를 사용하고
근본 원리에 대해 고민하면
커다란 혁신이 가능하다

Thank You