[인사]
안녕하세요. 빅데이터 처리 1조 발표를 맡은 전성원입니다.

[주어진 문제]
주어진 과제는 Facebook의 친구 데이터를 이용하여 공통된 친구를 가진 비율이 threshold 이상인 아이디의 쌍을 추출해내는 것입니다. 여기서 저희는 단순한 모든 사용자 쌍이 아닌, “아직 친구가 아닌” 모든 쌍을 추출하는 것이 문제의 핵심이라고 생각했습니다.

[Similarity Join 설명]
이 문제를 수업시간에 공부한 알고리즘을 기반으로 해결하기 위해 가장 단순한 Similarity Join부터 살펴봤습니다.

우선 데이터에서 각 ID별로 친구리스트를 생성합니다.
다음으로 FlatMap()과 GroupByKey()를 활용하여 inverted list를 생성합니다.
이 다음 FlatMap()을 이용하여 조인을 하기 위한 후보 pair를 생성합니다. 키는 후보 pair, 값은 1로 하여 생성합니다. 값을 1로 했기때문에 각 후보 pair는 자동으로 count됩니다.
마지막으로 각 후보 pair의 overlap에 대한 threshold를 계산합니다.
Filter를 이용해 계산된 threshold보다 큰 overlap을 가진 후보 pair를 선택합니다.
(이 모든 과정을 구현한 코드는 옆에 보시는 것과 같습니다.

[문제점 언급]
하지만 한 사람이 유독 너무 많은 친구를 가진다면 inverted list를 계산하는 데 O(n^2) 시간이 걸리므로 이러한 경우 매우 오래 걸릴 수 있습니다.
또한 similarity join은 inverted list의 모든 element에 대해 모든 pair를 생성하므로 많은 중복을 발생시켜 메모리 효율을 떨어뜨립니다.

[해결을 위한 노력]
이와 같은 문제점 해결을 위해 교수님께서 수업시간에 소개해주신 논문과 관련된 다른 논문들을 참고했습니다.
해당 논문들에서는 다양한 filtering 기법을 활용한 similarity join 알고리즘들을 소개하고 있습니다.
가장 기본이 되는 filtering method는 length filtering, prefix filtering, positional filtering입니다.
——2’00”
첫번째로 Length filtering입니다.
Jaccard Similarity threshold가 t로 주어질 때, y길이가 x길이와 threshold를 곱한 것보다 크지 않을 때, 조인후보 (x,y)는 후보에서 제외하는 filtering 기법입니다.

다음으로 Prefix filtering 입니다.
토큰의 전체 집합 U에 대해, 토큰의 순서 집합인 O와 레코드 집합이 주어졌을 때, O에서 정의하고 있는 토큰의 순서에 따라 각 레코드의 토큰을 정렬합니다.
이때, record x의 p-prefix는 앞에서부터 p개의 토큰을 의미합니다.
Prefix filtering 원리는 레코드 x와 y에 대해, overlap(x, y)의 값이 (알파) 보다 클 때, x의 p-prefix와 y의 p-prefix는 적어도 한 개 이상의 공통토큰을 가져야한다입니다.

이때 한 레코드의 prefix길이는 결국 다른 레코드의 길이에 의해 정해집니다. 이를 해결하기 위해 각 레코드마다 최대 prefix길이를 정해둡입니다. 최대 prefix 길이 공식은 이와 같습니다. Prefix filtering원리를 통해 조인 가능성이 전혀 없는 후보 pair를 제외시켜 기존의 similarity join보다 더 적은 조인 후보 pair를 만듭니다. 하지만 이 원리만으로는 문제점을 모두 해결할 수 없어 다른 여러가지 filtering 기법을 사용합니다.

마지막으로 positional filtering입니다.
앞서 언급한 prefix filtering은 prefix 범위 내의 overlap이 (알파)를 넘기지 않더라도 조인후보 pair로 선택합니다. 이를 통해 similarity join 결과를 누락시키지 않는 장점이 있지만, 불필요한 조인 후보의 pair이 너무 증가할 수 있다는 문제가 있습니다. 이를 해결하기 위해 positional filtering을 사용할 수 있습니다. 각 레코드 공통토큰의 위로 overlap의 최대크기를 계산하고, 그 중 (알파)보다 overlap 최대크기가 작은 경우의 레코드를 조인후보 pair를 생성하기 전에 제외합니다. 이때 overlap 최대크기는 현재 overlap크기 + min(x,y의 공통토큰 이후의 토큰 개수)로 계산할 수 있습니다. 이를 통해 조인후보의 쌍을 굉장히 많이 줄일 수 있습니다.
——5’00”
저희 팀에서는 총 4가지의 similarity join 알고리즘을 살펴봤습니다.
첫번째로 prefix filtering, length filtering, positional filtering을 사용한 PPJoin입니다.

교수님이 수업시간에 심화로 설명해주셨던 Join이기도 합니다.
단순 similarity join보다는 세가지의 filtering 기법을 적용해 더 적은 수의 inverted list와 조인후보 pair를 생성합니다. 하지만 여전히 많은 조인 후보 pair를 만들기 때문에 이를 검증하는 시간이 많이 걸립니다.

이러한 PPJoin의 성능을 개선시키기 위해 새로운 filtering 알고리즘인 suffix filtering을 적용한 PPJoin+를 제안하기도 했습니다. PPJoin+는 조인 후보 pair의 수는 압도적으로 줄였지만, 조인 후보 pair를 뽑아내기 위한 과정은 PPJoin과 동일해 큰 성능개선은 되지 않았습니다.

다음으로는 동적 prefix filtering을 사용하여 inverted list의 길이를 줄이는 방법을 소개하겠습니다.
x의 i번째 토큰을 (x,i)라고 하고, (x, i)의 inverted list의 토큰 중 하나를 y라고 할 때, x와 y는 (x, i)를 공통토큰으로 가집니다. 이 때 (x, y)가 앞에 filtering 기법들을 사용하여 조인 후보 쌍이 될 수 없으면 y를 (x,i)의 inverted list에서 삭제합니다.
[그림 그리기]

MPJoin은 inverted list를 만들면서 y의 prefix 범위를 넘는 토큰을 inverted list에서 삭제하는 동적 prefix filtering을 사용했습니다.

즉, 공통 토큰이 y의 prefix 범위 내에 있고, positional filtering을 만족하면 후보 Pair로 생성합니다. 이러한 filtering 방식은 조인 후보 pair 수를 유의미하게 줄였습니다.

다음으로 설명드릴 APJoin은 MPJoin과 동일하게 동적 prefix filtering을 사용합니다. 하지만 y뿐만 아니라 매번 달라지는 x의 prefix 범위도 고려합니다.
APJoin에서 조인 후보 pair이 될 수 없는지를 판단하는 기준은 세가지 입니다.
첫번째는 레코드 (y,j)가 length filtering을 만족하지 않는 경우 토큰 (y,j)를 inverted index에서 삭제합니다.
두번째는 j가 y의 prefix 범위를 벗어나면 토큰 (y,j)를 inverted index에서 삭제합니다.
현재 inverted list는 “정렬된” 친구 리스트를 바탕으로 만들고 있기 때문에, 반드시 다음 레코드 x의 길이는 더 큽니다. 따라서 prefix는 반드시 줄어들게 됩니다. 결국 (y,j)토큰이 한번 prefix 범위를 벗어나면 다음 레코드 x에 대해서도 벗어나게 됩니다. 따라서 (y,j)토큰을 삭제해도 문제가 되지 않습니다.
세번째는 i가 x의 prefix 범위를 벗어나면 다음 토큰으로 이동합니다.
(X,i)토큰은 prefix범위를 벗어났으므로 더이상 비교하지 않아도 되지만, (x,i)토큰이 인덱스 토큰(y,j)에 대해서는 prefix 범위 내에 위치할 수도 있으므로 삭제하지 않고 넘어갑니다.

APJoin 각 레코드에 대해 조인 후보 pair가 될 수 없는 토큰을 모두 걸러낸 inverted list를 생성하지만, 실제 조인 후보 pair는 다음의 방식으로 만들어집니다.
조인후보 pair는 VerifyZip에서 만들어지는데, positioal filtering을 활용해 공통토큰을 빠르게 계산하여 조인 후보 pair를 검증합니다.
이 리스트의 값이 (알파)보다 크면 조인 pair으로 추가합니다.

APJoin의 수도코드를 보시겠습니다.
우선 조인결과pair를 저장하는 리스트 S, inverted list I를 선언합니다.

y의 overlap score를 저장할 A리스트를 선언하고 다음 for문에서 각 레코드 별로 알파와 prefix 범위를 계산해서 저장합니다.

다음으로 동적 Prefix filtering 과정입니다. 앞서 설명드린 조인 후보 pair가 없는 세가지 조건의 수도코드입니다. 이 세가지 조건에 해당하지 않는 경우 조인 후보 Pair가 될 수 있으므로 A리스트의 값을 증가시킵니다.

Prefix filtering을 마친 후 inverted list를 생성합니다.

마지막으로 조인 후보 pair를 검증하고 출력합니다.

설명드린 4개의 similarity join의 성능을 비교했습니다.
첫번째 컬럼은 조인 후보 pair를 추출해내는 비교연산의 수이고, 두번째는 추출된 조인 후보 pair 수입니다. 세번째는 결과 pair 수 이고, 네번째는 실행시간을 나타냅니다.
PPJoin과 PPJoin+는 비교연산의 수는 동일하지만 추출된 조인 후보 pair 수가 크게 차이납니다. 조인 후보 수가 크게 줄어 실행시간 역시 줄었지만 큰 차이를 보여주진 못했습니다.

MPJoin과 APJoin은 조인 후보 pair 수 자체는 PPJoin과 큰 차이가 없지만 비교연산 수가 PPJoin보다 각각 절반, 1/4정도로 줄었습니다. 비교연산이 크게 줄자 실행 시간 역시 크게 줄은 것을 확인할 수 있습니다.

이러한 이유로 가장 성능개선이 잘 된 APJoin을 활용했습니다.
——10’00”
[코드 설명:3분]
우선 주어진 데이터로부터 친구리스트를 작성하고 정렬합니다.
정렬된 친구리스트를 R이라는 리스트로 저장합니다.

필요한 리스트들을 선언하고, 각 레코드별로 x와 y의 prefix 범위와 y의 overlap score를 저장할 A리스트를 초기화하고 다음 for문에서 각 레코드 별로 overlap의 threshold인 알파값과 prefix 범위를 계산해서 저장합니다.

다음으로 동적 Prefix filtering 과정입니다. 이부분은 앞서 보여드린 APJoin 수도코드를 기반으로 작성하였습니다. Prefix filtering을 마친 후 inverted list를 생성합니다.

마지막으로 조인 후보 pair를 검증하고 출력합니다.
이 과정에서 y의 overlap score가 0인 레코드는 넘어갑니다.
나머지 0이 아닌 것들에서 x와 y의 공통 토큰의 개수를 셉니다. 이 과정에서 positional filtering을 만족하지 않으면 조인 후보pair에서 제거합니다.
이렇게 검증한 조인 후보pair가 이미 친구인 경우를 확인하고, 친구가 아니면 조인 결과 리스트 S에 추가합니다.

[그림그리기]
이미 친구인 pair를 판단하는 방법은 다음과 같습니다. (x,y)pair 라고 할 때, x의 inverted list에 y가 있는지 확인합니다. 이 때 존재하면 이미 친구이므로 존재하지 않는 경우에만 S에 추가합니다.

결과를 출력하고, 주어진 threshold 별 결과 pair 수와 실행시간을 비교해보았습니다.
그래프를 보시면 threshold가 커질 수록, 결과 pair 수와 실행시간 모두 급감하는 것을 알 수 있습니다.
——13’00”
다음으로 기발하고 재미있는 아이디어를 추가적으로 적용해보고 싶어 다양한 친구추천 알고리즘에 대해 조사했습니다.

총 4개의 알고리즘을 살펴봤습니다, 결과적으로 해당 알고리즘을 활용하지는 않았기 때문에 간단히 설명하고 넘어가겠습니다.

첫번째로 Adamic Adar(아담 아다르) 알고리즘 입니다. 이 알고리즘은 공유하고 있는 이웃을 기반으로 각 노드의 근접성을 계산합니다. 노드 A와 B가 공유하고 있는 친구가 많더라도 그 친구들의 거리가 멀면 A와 B가 서로 친구가 될 가능성이 적다고 평가하는 알고리즘 입니다.
두번째는 Common Neighbors 알고리즘입니다. 연결되어있지 않은 두 노드들 간에 공통의 노드들이 공유된다면, 이 둘은 이후에 연결될 가능성이 높다는 원리를 이용합니다.
세번째로는 Friend of Friend Algorithm(FOAF) 알고리즘 입니다. 연결길이 l을 사용해서 한 개체와 거리가 l인 다른 개체를 그룹으로 형성합니다. 이렇게 개체들의 연결망을 형성하여 친구를 추천하는 알고리즘 입니다.
마지막으로 simRank 알고리즘입니다. SimRank는 “비슷한 사람에 의해서 가리켜지면, 비슷한 사람일 것이다”라는 가정에 기반한 node, similarity 알고리즘 입니다. 이를 계산하는 과정이 다음주에 배우게 될 Pagerank와 매우 유사합니다.

이상으로 빅데이터처리 PBL1 1조의 발표를 마칩니다. 감사합니다.
——15’00”

Q1. MPJoin의 |Comp|값이 PPJoin에 비해서 작은 이유?
인덱싱 토큰의 위치 j가 prefix 범위 보다 크면 삭제되기 때문

Q2 . APJoin의 |Comp|가 더 작은 것은?
탐색 레코드의 토큰 위치와 인덱싱 레코드의 토큰 위치가 설정된 prefix 범위 이내에서만 유사도 조인 후보쌍을 결정하기 때문.
→ Fig. 3과 Fig. 4는 Jaccard Similarity 한계치 t가 변화할 때 주어진 입력 데이터에 따라 각 알고리즘의 실행 시간을 보여준다.

맞춤 prefix filtering)
→ 탐색 레코드 x의 i번째 토큰 (x, i)와 이 토큰의 inverted index에서 가져온
인덱싱 레코드 y의 j번째 토큰 (y, j)를 비교하여 유사도 조인 후보 쌍으로 결정하는 과정

Fig. 1에서 빗금 친 칸들로 탐색 레코드와 인덱싱 레코드의 prefix의 범위를 설명
Fig. 2의 알고리즘에서 해당 경우의 prefix의 값으로 prefix_x[1:3] = {2, 2, 3}과 prefix_y[1:3] = {2, 1, 1}로 계산됨.
Table1에서 봤을 때, t = 0.8이고 탐색 레코드의 크기 |x| = 12일 때, 고려될 수 있는 인덱싱 레코드의 크기는 ⌈t * |x| ≤|y| ≤ |x|가 되어 |y|= 10, 11, 12가 된다.
(레코드의 비교 토큰들의 위치가 주어진 prefix 범위를 벗어나면 유사도 조인 후보에서 제외 된다.)
Table 1의 |y| = 12인 인 마지막 네 열에서는 크기가 12인 탐색 레코드 x가 다음 처리 과정에서 인덱싱 레코드 y로 사용되는 경우에 해당되는 변수들의 값을 설명하고 있다.
prefix_y의 값들 중에서 최대값은 2가 되어 |x| = 12일 때 max_index_prefix = 2이므로 서로 같음을 보여준다.
→ 이것은 크기가 12인 탐색 레코드 x의 토큰들 중에서 앞에서부터 Equation (6)의 max_index_prefix 만큼만 inverted index에 저장하면 충분하다는 것을 의미한다.

APJoin 알고리즘
- PPJoin & MPJoin에서 사용한 prefix filtering과 positional filtering을 적용
- MP Join에서 사용한 동적으로 갱신되는 인덱싱 레코드의 prefix_size 적용하는 방법 → 탐색 & 인덱싱 레코드의 prefix 범위 설정에 이용함.

한 레코드가 탐색 레코드로 처음 고려될 때,
이 레코드의 prefix의 범위를 Equation (3)으로부터 설정함.
이 레코드와 prefix 범위 안에 들어올 수 있는 인덱싱 레코드의 prefix의 범위를
Equation (4)로부터 맞춤 방식으로 초기에 설정함.

- 탐색 레코드와 인덱싱 레코드의 prefix 범위가 설정되어 있기 때문에
비교되는 탐색 토큰과 인덱싱 토큰의 두 레코드가 설정된 prefix 범위 안에 있으면
유사도 조인 후보 쌍으로 결정함.

AP JOIN 수도코드)
입력 : 레코드들의 집합 R과 주어진 쟈카드 유사도 한계치 t
출력 : 한계치 t 이상의 값을 갖는 모든 레코드 쌍
—> 각 레코드는 토큰들로 구성되고, 레코드들은 토큰들의 개수에 따라
비내림 차순으로 정렬되어 있다.
코드 설명)
단계 5 - 10에서 Equations (2) - (6)에 해당되는 변수들을 계산한다.
단계 8에서 탐색 레코드 x가 주어지면 인덱싱 레코드 y의 크기가 결정되어 다른 식들에 사용된다.
단계 14 - 21은 탐색 토 큰 (x, i)와 인덱싱 토큰 (y, j)로 앞에서 설명한 조건 검사 만으로 유사도 조인 후보 쌍을 결정한다.
조건 검사에 맞지 않는 현재 토큰이 유사도 조인 후보가 될 가능성이 전혀 없으면 inverted index에서 삭제하여 이후 비교 검사되는 인덱싱 레코드의 토큰들의 개수를 줄인다.
단계 18의 조건은 Fig. 1의 (a)와 (b)의 탐색 레코드 x의 3번째 토큰과 같이 현재 인덱싱 레코드의 토큰과 조인 후보가 될 가능성이 없으면 현재 토큰은 처리하지 않고 다음 토큰으로 이동한다. x의 3번째 토큰은 Fig. 1의 (c)에서 사용될 것이다.
단계 21에서 탐색 토큰 (x, i)와 인덱싱 토큰 (y, j)가 두 레코드 x 와 y의 prefix 범위 안에 들어오기 때문에 x와 y가 유사도 조인 후보 쌍으로 생성된다.
단계 22 - 24에서 앞으로 사용할 토큰들을 inverted index에 저장한다.
단계 25의 함수 VerifyZip은 지퍼 합병 방식으로 공통부분을 계산하여 유사도 조인 쌍을 결정하는 함수다.
이 함수에서 두 레코드의 토큰들을 비교하면서 현재 공통부분 토큰들의 개수, 남아있는 토큰들의 개수, 알파 값을 비교하여 계속 비교 검증 할 것인 지 또는 중단할 것인 지를 결정하여 빠르게 유사도 조인 후보 쌍을 검증한다.

[실험 분석 결과]

- 표 설명
  |Join|은 유사도 조인 결과 쌍들의 개수이고,
  |Cand|는 유사도 조인 후보 쌍들의 개수이고, Time은 실행시간을 나타낸다.
  |Comp|는 탐색 토큰과 인덱싱 토큰을 검사하여 유사도 조인 후보 쌍을 결정하는 단계들의 회수를 나타내는 값이다.
- 성능 분석 결과
  알고리즘의 성능 평가를 위해 동일한 시간 복잡도의 범주를 갖는 4개의 알고리즘들을 프로그래밍하여 DBLP 데이터를 입력으로 한 실험결과를 Table 2에서 보여주고 있다.
  → 실행 시간 비교에서 본 논문에서 제안된 APJoin은 실행속도면 에서 PPJoin에 비해 68% 그리고 MPJoin에 비해 32%의 성능 개선을 보여줌.

[논문 결론] :발표 안하고 넘어가도 되지 않을까요?
본 논문에서 각 레코드가 토큰들로 이루어진 대용량의 레코드들 중 서로 유사한 레코드들을 찾아내는 효과적인 유사도 조인(similarity join) 알고리즘을 제안한다.
- 이 알고리즘은 접두 필터링(prefix filtering) 원리에 따라 탐색 레코드와 인덱싱 레코드의 접두 토큰들의 개수를 설정하여 이 범위 내에 들어오는 토큰이 일치하면 유사도 조인 후보로 만들어 검증한다.
- 실제 데이터를 사용한 실험을 통하여 제안된 알고리즘이 기존의 접두 필터링에 기반한 알고리즘들에 비해서 더 빠르게 결과를 찾아내는 것을 보여주었다. 제안된 알고리즘은 대용량 데이터 분석 등에 활용될 수 있다.
→ APJoin 알고리즘은 탐색 레코드의 토큰 위치와 인덱싱 레코드의 토큰 위치가 설정된 prefix 범위 이내에서만 유사도 조인 후보쌍을 결정하기 때문에 실험에서도 찾는데 빠른 시간이 걸리므로 효율적이다.

APJoin 알고리즘에서 개선해야 될 부분은 Table 2의 |Cand|로 표시된 유사도 조인 후보 쌍들의 개수를 줄이는 것과 ENRON 데이터와 같이 평균 토큰 개수가 큰 레코드들을 처리하는 시간을 줄이는 것으로 특히, APJoin 알고리즘 에서 PPJoin+ 알고리즘과 같이 후보 수를 줄이는 방법을 모색하면 더 좋은 성능을 갖는 알고리즘이 될 것이다.