1
안녕하세요. PBL2 1조 발표를 맡은 전성원입니다.

2
PBL2는 페이지랭크 알고리즘의 계산속도를 높이기 위해 블럭기반으로 구현하는 것입니다.
저희는 페이지랭크 알고리즘을 다시한번 살펴보고, 블럭기반 구현을 위해 수업시간에 살펴본 논문을 다시한번 살펴보았습니다. 이를 바탕으로 코드를 우선 구현하고, 성능을 평가하였습니다. 마지막으로 더 나은 결과를 위한 추가적인 아이디어를 조사했습니다.

3
페이지랭크 알고리즘은 구글검색엔진의 기반 알고리즘으로, 하이퍼링크를 이용해 웹 페이지 중요도를 측정합니다.
페이지u가 페이지v 에서 언급된다는 것은 페이지v의 저자가 페이지u를 중요하게 생각하고 있음을 내포합니다.
또한 어떤 웹페이지가 중요하다는 것은 다른 웹페이지들로부터 많이 언급되거나, 많이 언급되는 페이지가 언급하는 경우를 의미합니다.

4
페이지랭크 알고리즘에서는 이러한 중요도를 측정하기 위해 웹페이지를 노드로, 하이퍼링크를 엣지로 하여 웹을 directed graph로 표현합니다. 중요성은 다음과 같은 공식을 통해 도출됩니다. Nu를 페이지 u의 outdegree, Rank(p)를 페이지p의 중요성이라고 한다면, link(u,v)는 페이지 v에 대해 Rank(u)/Nu만큼 중요합니다.
이때 Bv는 페이지 v를 가리키고 있는 페이지 집합입니다. 이 공식을 이용해서 Rank 초기값들을 반복하여 갱신하고, 갱신되는 값의 차이가 threshold 범위 이하라면 안정된 것으로 판단하고 그 때의 Rank값을 사용합니다.

5
기존에 실습과제로 주어졌던 PankRank구현은 link의 수가 크게 증가했을 때 계산이 되지 않았습니다. 따라서 블럭기반으로 구현하여 계산속도를 높이고자함이 PBL2의 핵심과제 입니다. 이를 위해 수업시간에 소개해주신 논문을 참고하였습니다.

6
논문에서 소개하고 있는 블럭기반 PageRank 구현은 다음과 같습니다.
가장 먼저 전처리 단계로, 자식 페이지가 없는 dangling 페이지, 즉 outdegree가 0인 페이지들을 제거합니다. 데이터로 주어지는 원본 그래프는 dangling 노드들이 줄줄이 연결된 경우가 많기 때문에 제거한 그래프에서 한번 더 제거합니다. 두번째로 노드 ID를 0부터 차례대로 매깁니다. 세번째로 각 노드에 대해 ID, outdegree, 가리키고 있는 노드들의 ID집합을 디스크에 저장합니다. 마지막으로 source의 Rank 백터와 source-dest로 이루어진 2차원 행렬을 곱해 Dest Vector를 계산합니다.

7
마지막 과정인 Dest vector 계산하는 부분을 블럭기반으로 구현하였습니다
8
이 때 블럭을 구분하는 기준에 따라 계산량이 달라지기 때문에, Dest 노드의 번호가 작은 순으로 정렬하고 3개의 구간으로 나누어 Link 파일은 분할해 계산하고 합쳤습니다.

9
계산 속도를 올리기 위한 아이디어 중 하나로 코어 수를 늘려보았습니다. 스파크에서 코어수를 늘리기 위해 다음과 같이 설정하였고, 싱글코어, 듀얼코어, 쿼드코어를 이용하여 실행시간을 비교해보았습니다.

10
제시된 데이터 셋을 source 와 destination set의 쌍으로 매핑하였습니다.
기존에 구현했던 PageRank에서 블록 개수를 지정하고 변경된 수렴조건을 수정하였습니다. 블록개수 또한 다양하게 설정하여 실행시간을 비교하였습니다.

11
기존에는 전체 데이터에 대해 source, destination pair쌍을 생성했지만, 블럭 단위로 link structure를 우선 구축한 후, block id와 source/destination쌍을 매핑하는 함수와 블록 별로 PageRank를 계산하는 함수를 작성했습니다.

12
제시된 데이터 셋을 source와 destination set으로 매핑한 뒤 실행과정을 말씀드리겠습니다.
첫번째로 destination node id의 범위에 따라 destination set을 블록 개수만큼 나눠서 block id를 붙입니다.
다음으로 같은 block id를 가지는 (source, destination set조각)을 groupbykey로 모아 블록을 만듭니다.
maxIter만큼 반복하며 블록 별로 PageRank 값을 계산합니다.
마지막으로 블록 별로 계산한 PageRank 값 리스트를 reduce로 합칩니다.

13
코드를 실행하기 전에 코어수와 블록수가 변경됨에 따른 변화를 예측해 보았습니다. 코어는 일꾼으로 비유할 수 있습니다. 따라서 코어 수가 많다면 더 빠른 속도로 작업을 할 수 있을 것이라고 예측하였습니다.
하지만 코어 수가 작업 수보다 많아지면 남는 코어 수가 많아지는 것 뿐이기 때문에 실행시간이 줄어들 지 않을 것이라고 생각했습니다.

14
블록의 경우 해야하는 일이라고 비유할 수 있습니다. 따라서 블록의 수가 코어의 수에 비해 지나치게 많으면 일 처리 속도가 느려 진다고 예측하였습니다.

15
저희는 수업시간에 배운 알고리즘을 구현하는 것을 기반으로 총 3번의 시도를 통해 최종 결과를 도출했습니다.
첫번재 시도와 두번째 시도에서는 모두 source 와 destination의 pair 쌍을 생성한 뒤 destination 범위에 따라 block을 나눴습니다. 블럭을 만들기 전 데이터 개수는 총 Edge 개수만큼이므로 블럭으로 구현했을 때의 효율이 발생하지 않았습니다.
이렇게 블럭 수만큼 RDD를 생성한 뒤에 첫번째 시도에서는 groupbykey를 이용하여 블럭 내에서 pagerank를 계산하고 합쳤습니다. 이에 대한 성능 개선을 위해 두번째 시도에서는 같은 키를 가진 페어를 같은 파티션에 넣어 파티션 사이의 데이터 셔플링을 줄여보고자 했습니다.
마지막으로 세번째 시도에서는 첫번째와 두번째 방법에서 블럭을 나누는 과정에 문제가 있음을 알고 이를 개선하였습니다. 원래의 link structure에서 source와 destination의 pair를 생성하지 않고, 블럭 수 만큼 각 link structure를 쪼개서 블럭으로 먼저 나눠 Block id를 키로 하고, 쪼갠 link structure로 매핑합니다. 그다음 Groupbykey로 블럭을 모아 블럭 내에서 pagerank를 계산하고 합쳤습니다. 세번째 시도에서 RDD의 수는 블럭 수 만큼이라 첫번째 시도와 동일하지만 블럭을 만들기 전 데이터의 개수는 link structure의 개수로 첫번째 시도보다 데이터 개수가 압도적으로 적습니다. 데이터 개수가 적기 때문에 계산 복잡도 역시 크게 줄어 큰 성능향상이 있었습니다.

16-22
세가지 시도 모두의 실제결과는 예측과 많이 달랐습니다.
첫번째 시도는 코어 수 별로 5번씩 실행해 실행시간의 평균을 살펴보았습니다. 평균적으로 27초에서 30초 정도가 걸렸습니다. 코어 가 하나일 때, 두개일 때, 네개일 때 모두 블럭 수가 4개일 때 가장 짧은 실행시간을 보이고, 나머지 블럭 수에서는 일정하지 않은 결과를 보였습니다.
두번째 시도는 평균적으로 80초 정도 소요되었고, 가장 많은 시간이 소요된 알고리즘입니다. 블록 수, 즉 파티션의 수가 늘어날 수록 실행시간도 증가하였습니다.
마지막 세번째 시도는 코어 수 별로 1번 씩 실행한 결과를 살펴보았습니다. 평균적으로 8초 정도가 소요되었고, 코어 수 별로 2초정도의 시간이 줄어드는 모습을 보여 각 코어가 효율적으로 태스크를 수행하고 있음을 알 수 있었습니다.

23
저희팀의 결과를 종합해보자면, 논문을 바탕으로 첫번째 시도를 구현했습니다.
하지만 첫번째 시도에서 성능이 예상한 것보다 좋지 않아 관련 자료를 찾아보던 중, groupbykey가 shuffle 측면에서 굉장히 비효율적이라는 것을 알게 되었습니다. 그래서 두번째 시도에서는 같은 키를 같은 파티션에 넣고 Partitionby를 사용했지만 성능은 더욱 좋지 않았습니다. 따라서 마지막에는 블록을 만드는데 사용되는 데이터 수를 줄이는 방향으로 세번째 시도를 했습니다. 이때 pair쌍이 아닌 link structure를 사용해서 블럭을 생성한 결과가 가장 좋았습니다.

24
앞서 말씀드렸던 성능개선을 위한 관련자료 조사를 하면서 알게된 점에 대해 말씀드리겠습니다.

우선 파티션과 코어의 관계입니다. 하나의 파티션은 하나의 코어가 맡기 때문에 결국 파티션의 크기가 코어 당 필요한 메모리 크키를 결정합니다. 따라서 파티션의 크기와 개수가 스파크 성능에 큰 영향을 미치게 되고, 불필요하게 많은 파티션을 할당하면 오히려 더 오래 걸릴 수 있습니다.

25
좀 더 자세히 설명하면, 스파크 드라이브는 모든 파티션에 대한 메타데이터를 보관하고, 각 파티션에 대한 스케쥴링을 해야합니다. 그러므로 너무 많은 파티션의 수는 많은 오버헤드를 발생시킬 수 있어 좋지 않습니다. 그 와 반대로 파티션의 수보다 코어 수가 많은 경우에는, 일부 코어가 작업을 하지 않고 쉬게 되므로 효율적이지 않습니다. 따라서 적절한 파티션의 수와 코어 수를 선택해야 합니다.

26
다음으로 저희가 알게된 점은 서버코어 수보다 많은 실행코어를 할당할 수 없다는 점입니다. 따라서 스파크를 이용하여 알고리즘을 구현할 때, 서버코어 수를 반드시 확인해야 합니다.
저희가 사용한 VM의 기본 코어 수는 1개 이기 때문에 이번 과제를 진행할 때 변경하고 진행했습니다.

27
마지막으로 알게된 점입니다. Map, Key By 등으로 새로운 RDD를 생성할 때는 파티션이 변경되지 않습니다. 따라서 각각의 파티션들이 키와 상관없이 데이터를 갖고 있을 때, Key-Value로 이루어진 pair RDD를 생성하고 Key를 기반으로 한 연산을 진행하게 될 경우, 많은 셔플링이 발생하게 됩니다. 따라서 같은 키를 가진 데이터들을 같은 파티션에 모이도록 할 경우 성능을 크게 개선할 수 있습니다.

[추가 아이디어]

[BlockRank]Exploiting the Block Structure of the Web for Computing PageRank

1. 하이퍼링크의 대부분은 intra-host/domain link
* intra-host link: 같은 호스트 내에서 연결된 링크
www.naver.com/111/2222. <-> www.naver.com/22/333/444
* inter-host link: 다른 호스트끼리 연결된 링크
www.naver.com/22/333 <-> www.hanyang.ac.kr/111/22

-보다시피 intra 링크는 비슷한 주소 구조를 가진다. 따라서 이러한 nested block 구조는 자연스럽게 링크 그래프로 정렬된다.

2. The GeoCites Effect
=친구추천 알고리즘에서 친구가 많은 노드와 같은 효과
작은 호스트들이 제거되면, 큰 호스트는 높은 intra호스트 밀도를 같게된다. 이때 매우 적은 호스트들만 GeoCites 효과를 겪는다.

3. BlockRank

1)도메인에 따라 웹을 블럭으로 나눔
2)각 블럭에 대한 로컬 페이지랭크를 계산한다.
3)각 블럭에 대한 블럭랭크, 즉 상대적 중요성을 평가한다.
4)블럭랭크값을 이용해서 블럭 내의 로컬 페이지랭크에 가중치를 부여하고, 이를 통해 전체 페이지 랭크 벡터를 추정한다.
5)이 추정된 페이지랭크 벡터를 초기값으로 사용한다.

4. 주요장점
Caching effect에 대한 성능향상
로컬 페이지랭크 벡터는 빨리 수렴한다.
2)의 로컬 페이지랭크 계산은 완전히 병렬적으로 이뤄지며, 재활용이 가능하다.

지역참조를 감소시킴
계산복잡도 감소
높은 병렬 계산성
계산결과 재활용 가능: 개인화된 PageRank를 전체 PageRank 계산에 활용