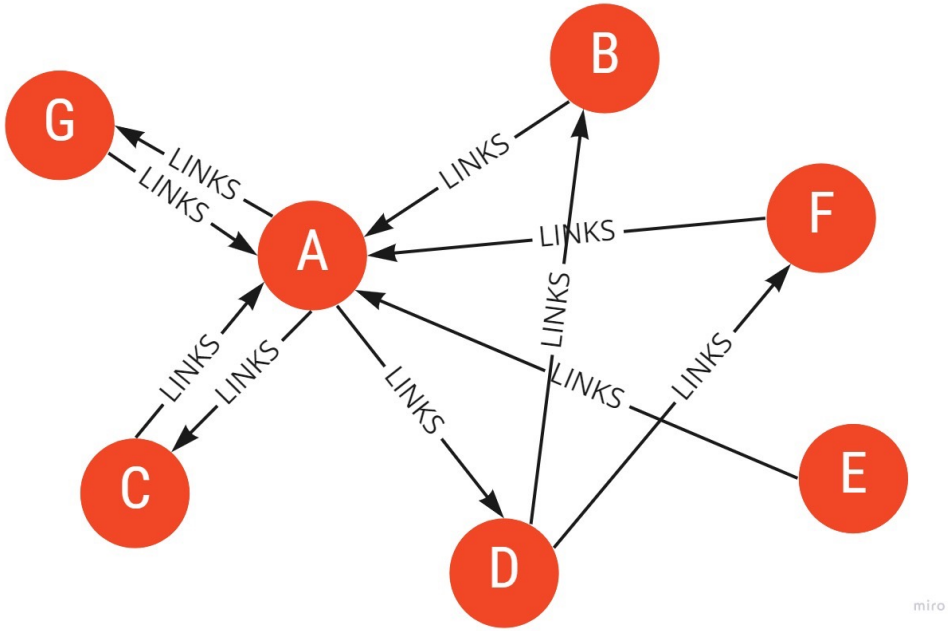
PBL-2

Efficient Computation of PageRank



1조

신민경 이정인 전성원

Content

- PageRank Algorithm
- Block Based Strategy
- Efficient PageRank Using Spark
- Team Analysis & Conclusion
- More idea

PageRank Algorithm

✓ Google 검색 엔진의 기반 알고리즘으로, 하이퍼링크를 이용해 웹 페이지 중요도를 측정한다.

✓ 페이지 u가 페이지 v의 링크를 참조하고 있다면,
페이지 u의 저자는 페이지 v의 중요성을 함축적으로 내포한다.

✓ 핵심 아이디어 : 링크 정보를 이용하여 노드(페이지)마다 점수(순위)를 매기는 것

그렇다면 **어떤 노드에 높은 점수를 줄 것인가?**

1) 많은 링크를 받는 페이지 → 높은 점수

2) 높은 점수의 페이지로부터 링크를 받으면 → 높은 점수

PageRank Algorithm

✓ **하이퍼링크 네트워크 :** Web을 directed graph로 보는 형태이다.

Node : 웹 페이지 , Edge : 하이퍼링크

✓ **중요성을 얼만큼 내포하고 있을까 ?**

Nu를 페이지 u의 outdegree , Rank(p)를 페이지 p의 중요성이라고 하자.

그렇다면 link (u, v)는 페이지 v에 대해 Rank(u) / Nu 만큼 중요하다.

$$\forall_v Rank_{i+1}(v) = \sum_{u \in B_v} Rank_i(u)/N_u$$

Bv는 페이지 v를 가리키고 있는 페이지 집합이다. 위와 같이 페이지들의 Rank를 반복하여 갱신하고,

이전 단계에서 계산한 값과 차가 threshold 범위 이하라면 안정된 것으로 판단하여 종료한다.

Block-Based Strategy

✓ Efficient Computation of PageRank 논문 참고

Efficient Computation of PageRank

Taher H. Haveliwala*
Stanford University
taherh@db.stanford.edu

October 18, 1999

Abstract

This paper discusses efficient techniques for computing PageRank, a ranking metric for hypertext documents. We show that PageRank can be computed for very large subgraphs of the web (up to hundreds of millions of nodes) on machines with limited main memory. Running-time measurements on various memory configurations are presented for PageRank computation over the 24-million-page Stanford WebBase archive. We discuss several methods for analyzing the convergence of PageRank based on the induced ordering of the pages. We present convergence results helpful for determining the number of iterations necessary to achieve a useful PageRank assignment, both in the absence and presence of search queries.

Naive Technique

Step of Technique

I. 가장 먼저 전처리 단계로 자식 페이지가 없는 dangling 페이지(outdegree=0)들을 제거한다.

원본 그래프는 dangling 노드들이 줄줄이 연결된 경우가 많기 때문에 제거한 그래프에서 한 번 더 제거한다.

I. 노드 ID가 0부터 시작하여 차례대로 매겨진다.

II. 각 노드에 대하여 ID , OutDegree , 가리키고 있는 노드의 ID 집합들을 디스크에 저장한다.

III. Source의 Rank 벡터와 source-dest로 이루어진 2차원 행렬을 곱하여 Dest vector를 계산한다.

Block-Based Strategy

✓ 아이디어 : (Dest) 벡터를 분할하여 계산하고 만들어진 (Dest) 벡터를 합친다.

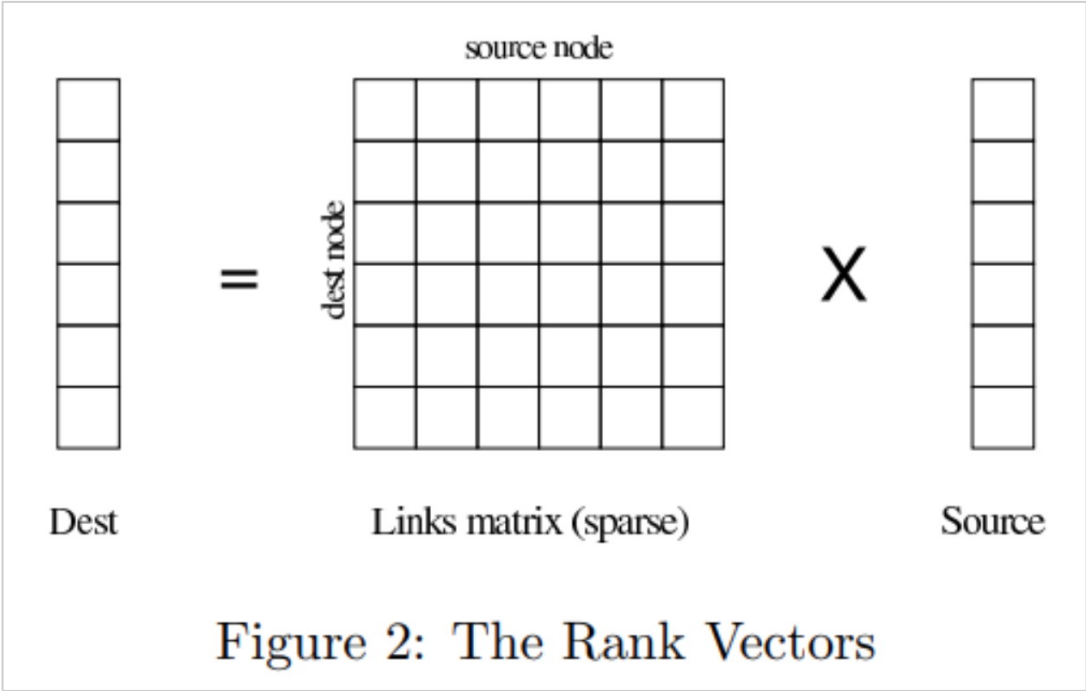


Figure 2: The Rank Vectors

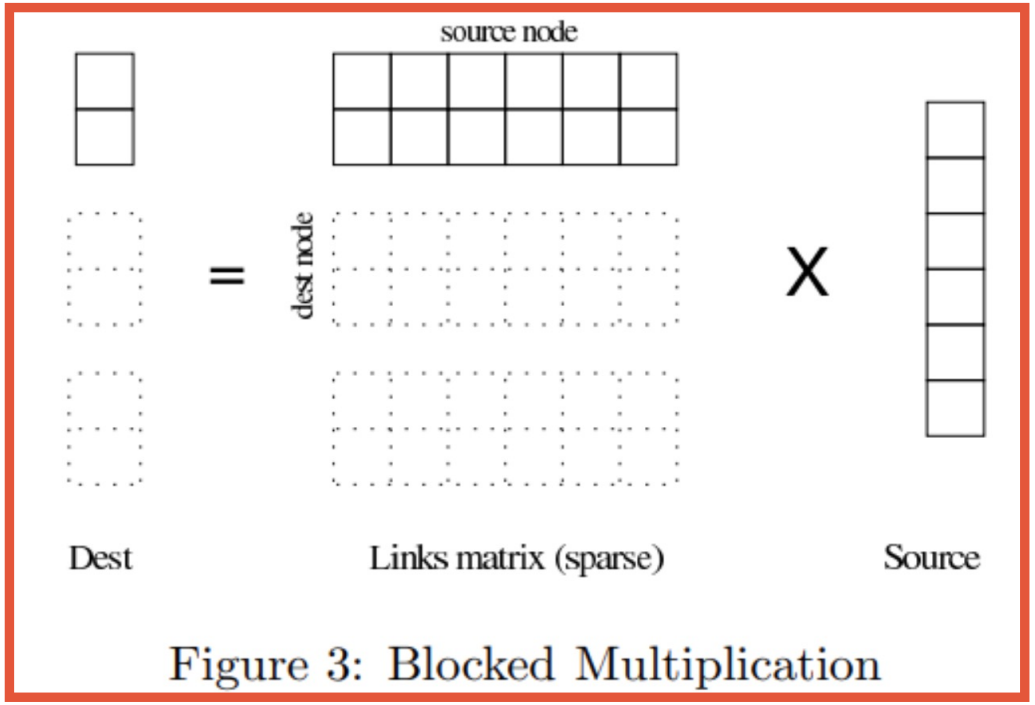


Figure 3: Blocked Multiplication

Block-Based Strategy

Source Node (32-bit id)	Out Degree (16-bit integer)	Destination Nodes (series of 32-bit id's)
0	4	12, 26, 58, 94
1	3	15, 56, 81
2	5	9, 10, 38, 45, 78

Figure 1: The Link Structure

Source Node (32-bit id)	Out Degree (16-bit integer)	Num Out (16-bit integer)	Destination Nodes (series of 32-bit id's)
0	4	2	12, 26
1	3	1	15
2	5	2	9, 10

Links Bucket 1 (0 <= dest < 32)

Source Node (32-bit id)	Out Degree (16-bit integer)	Num Out (16-bit integer)	Destination Nodes (series of 32-bit id's)
0	4	1	58
1	2	1	56
2	5	2	38, 45

Links Bucket 2 (32 <= dest < 64)

Source Node (32-bit id)	Out Degree (16-bit integer)	Num Out (16-bit integer)	Destination Nodes (series of 32-bit id's)
0	4	1	94
1	2	1	81
2	5	1	78

Links Bucket 3 (64 <= dest < 96)

Figure 4: Partitioned Link File

- Destination 노드의 번호가 작은 순으로 정렬하고 **3개의 구간으로 나누어 Link 파일을 분할한다.**
- **각 구간별로 Dest 벡터를 계산하고 합친다.**

Efficient Computation of PageRank

- 코어 수 설정

싱글 코어 설정

```
from pyspark import SparkContext, SparkConf

conf = (SparkConf()
         .setMaster("local")
         .setAppName("pbl_2")
         .set("spark.executor.cores", "1")) # 코어 수

sc = SparkContext(conf = conf)
```

멀티 코어 설정

```
from pyspark import SparkContext, SparkConf

conf = (SparkConf()
         .setMaster("local")
         .setAppName("pbl_2")
         .set("spark.executor.cores", "4")) # 코어 4개

sc = SparkContext(conf = conf)
```

Efficient Computation of PageRank

Source , Destination set 매핑

```
data = sc.textFile("test_dataset.txt").flatMap(lambda file: file.split('\n')).map(lambda line: line.split(' '))\
            .map(lambda line: (int(line[0]), [int(i) for i in line[1:]]))
N = data.count()

block = [16, 8, 4, 2, 1]
maxIter = 20
execution_time = []
```

1) 블록의 수를 16,8,4,2,1로 다르게 설정하고 수렴 조건인 maxIter을 20으로 설정한다.
2) 블록 안에 Source , Destination set 집합의 Pair 쌍을 생성한다.

Efficient Computation of PageRank

블록 단위로 Link Structure 구축 (block id, (source, destination set)) 매핑

```
def make_link_set(dest_set, block_cnt, block_size):
    source = dest_set[0]
    block_based_dest_set = [(source, []) for i in range(block_cnt)]

    for dest in dest_set[1]:
        block_based_dest_set[dest // block_size][1].append(dest)

    for i in range(block_cnt):
        if len(block_based_dest_set[i][1]) != 0:
            yield (i,block_based_dest_set[i])
```

블록 별 Page Rank 계산

```
def pr_map(block_data, block_size, N):
    block_id, link_sets = block_data
    new_sr = np.zeros(block_size) + 1.5/N

    for link_set in link_sets:
        source, dest_set = link_set
        val = pr.value[source] / len(dest_set)

        for dest in dest_set:
            new_sr[dest // block_size] += val

    yield new_sr.tolist()
```

Efficient Computation of PageRank

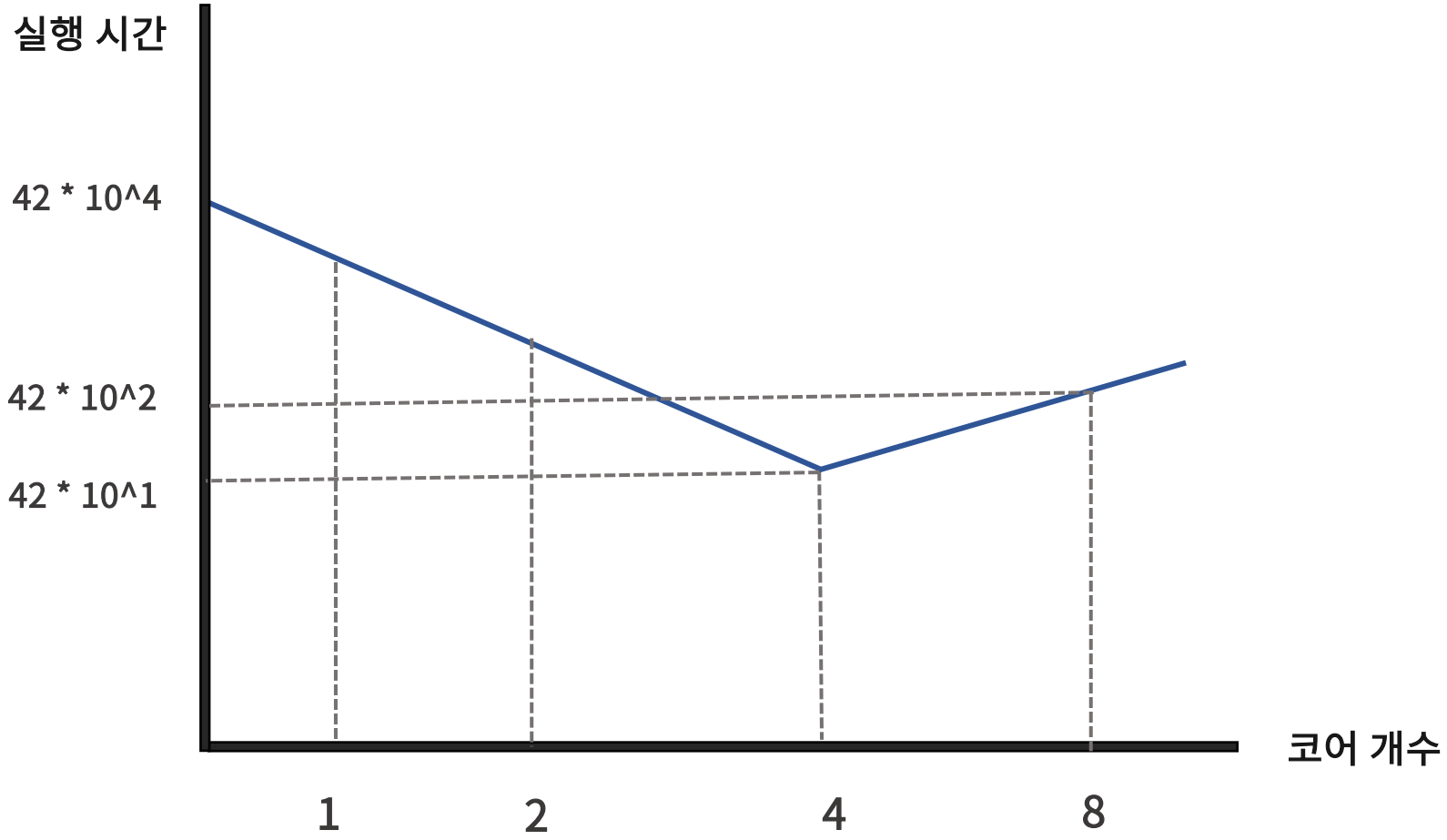
Block Based Page Rank Calculation (block_id, (source, destination set))

```
for block_cnt in block: # block의 개수를 바꿔가며 실행
    start = time.time()

    pr = sc.broadcast([ 1. / float(N) for _ in range(N) ])
    block_size = (N // block_cnt) if N % block_cnt != 0 else (N // block_cnt)+1

    # source별 destination set을 블록 구간에 따라 분리함
    block_based_set = data.flatMap(lambda s: make_link_set(s, block_cnt, block_size)).groupByKey()

    for _ in range(maxIter):
        itr = block_based_set.flatMap(lambda b: pr_map(b, block_size, N)).reduce(lambda x,y : x+y)
        pr = sc.broadcast(itr)

    end = time.time()
    execution_time.append(end-start)

    print("the number of blocks: {}".format(block_cnt))
    print("duration: {}".format(end-start))
    print()
```

1) destination node id의 범위에 따라 destination set을 블록 개수만큼 나누어 번호를 붙인다.
2) 같은 블록 id를 가지는 (source, destination set)을 groupByKey로 모아 블록을 만들어 준다.
3) maxIter 만큼 반복하며 블록 별로 PageRank 값을 계산한다.
4) 블록 별로 계산한 PageRank 리스트를 reduce로 합친다.

Prediction of Result

✓ 코어 수에 따른 결과 예측 (블록이 4개일 때 코어 수에 따른 변화)



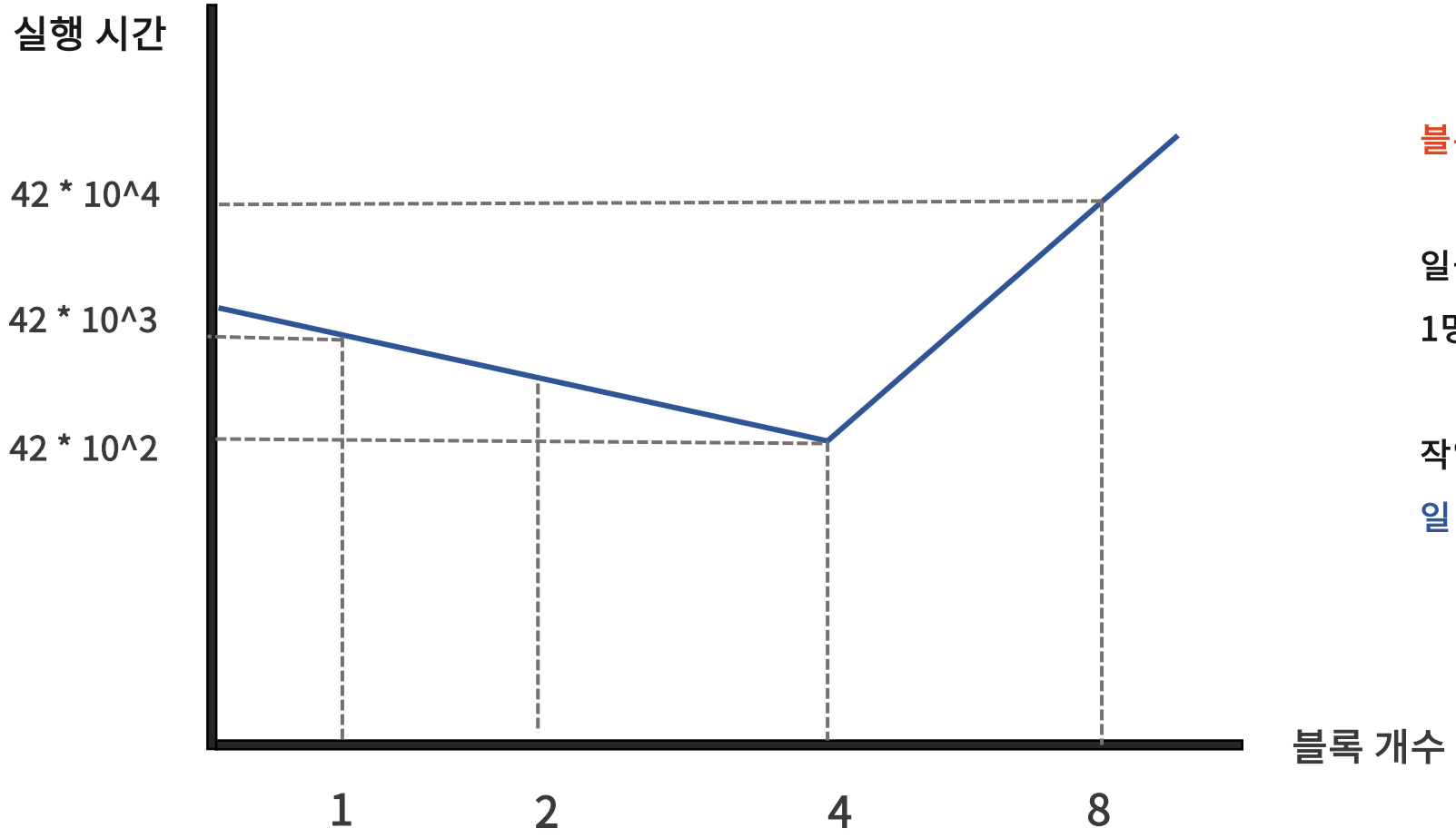
코어 = 일꾼으로 비유해보자.

일꾼이 1명보다 4명인 경우에
더 빠른 속도로 일을 처리할 수 있고

작업이 4개일 때 일꾼이 8명이면
한 사람당 하나의 작업만 할 수 있기에
변화가 없을 것이라고 예상함.

Prediction of Result

✓ 블록 수에 따른 결과 예측 (코어의 수가 4개일 때 블록 수에 따른 변화)



블록 = 작업으로 비유해보자.

일꾼이 4명일 때, 작업이 4개면
1명씩 일을 나눠 동시에 진행하면 되지만

작업이 8개면, 사람이 4명이나 부족하기에
일 처리 속도가 느려 진다고 예상함.

Experiment

1번째 시도

Block 안에

(Source, Dest) Pair 쌍

GroupbyKey

데이터 개수 = Edge개

2번째 시도

Block 안에

(Source, Dest) Pair 쌍

RDD 1개

Partitionby

데이터 개수 = Edge개

3번째 시도

Block 안에

Link structure

GroupbyKey

데이터 개수 =

Block 개수 x Node 개수

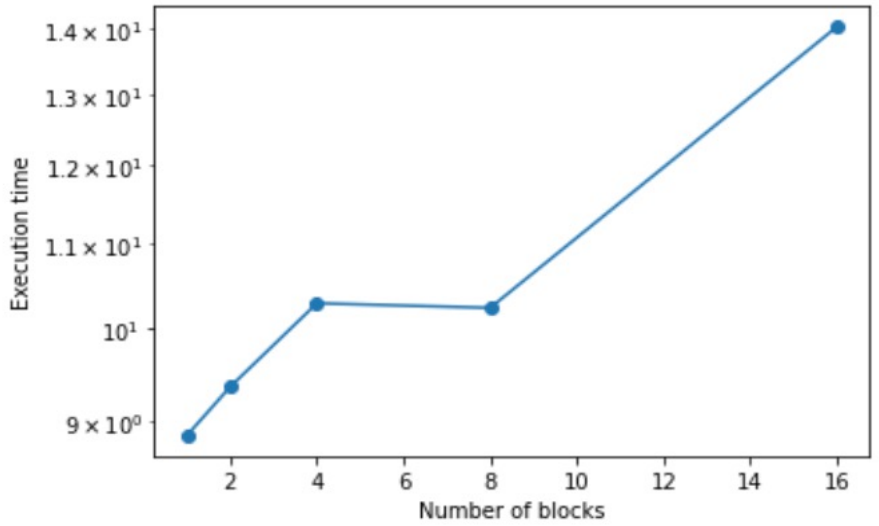
3 Experiment Result

✓ Core 1개인 경우 (직접 만든 데이터셋)

```
the number of blocks: 16
duration: 14.0278959274292

the number of blocks: 8
duration: 10.228794813156128

the number of blocks: 4
duration: 10.281517744064331

the number of blocks: 2
duration: 9.360647916793823

the number of blocks: 1
duration: 8.861914873123169
```

그래프

```
plt.plot(block, execution_time, marker = 'o')

plt.xlabel("Number of blocks")
plt.ylabel("Execution time")

plt.yscale('log')
plt.show()
```



코어 1개를 1번 실행시킨 결과

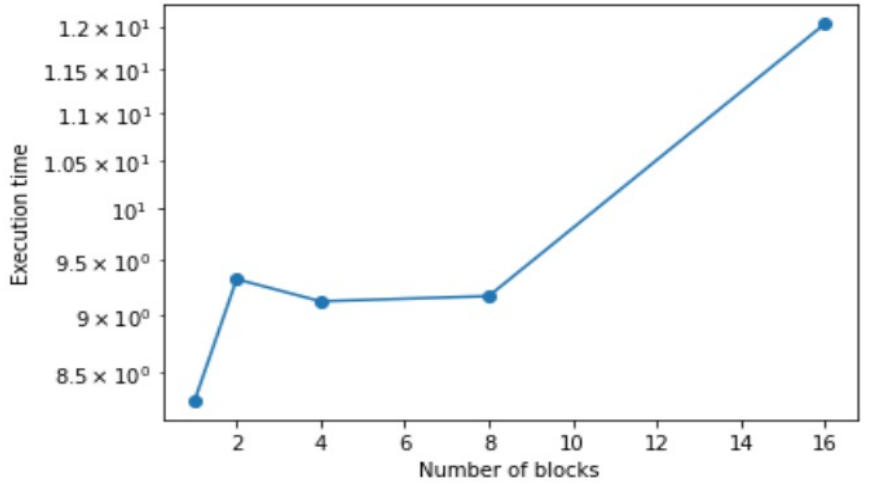
3 Experiment Result

✓ Core 2개인 경우 (직접 만든 데이터셋)

```
the number of blocks: 16
duration: 12.032623052597046

the number of blocks: 8
duration: 9.168304920196533

the number of blocks: 4
duration: 9.120410919189453

the number of blocks: 2
duration: 9.324203729629517

the number of blocks: 1
duration: 8.25999641418457
```

그래프

```
plt.plot(block, execution_time, marker = 'o')

plt.xlabel("Number of blocks")
plt.ylabel("Execution time")

plt.yscale('log')
plt.show()
```



코어 2개를 1번 실행시킨 결과

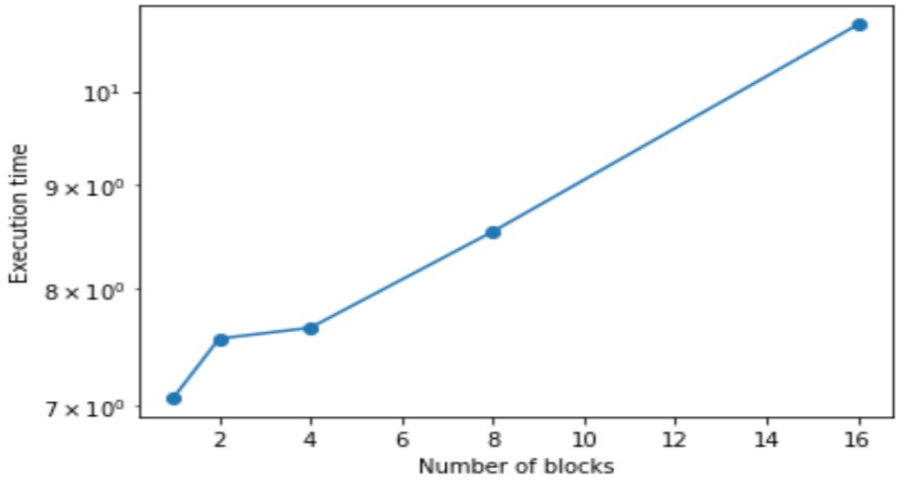
3 Experiment Result

✓ Core 4개인 경우 (직접 만든 데이터셋)

```
the number of blocks: 16
duration: 10.784173488616943

the number of blocks: 8
duration: 8.530853986740112

the number of blocks: 4
duration: 7.650415897369385

the number of blocks: 2
duration: 7.556777000427246

the number of blocks: 1
duration: 7.0680251121521
```

그래프

```
plt.plot(block, execution_time, marker = 'o')

plt.xlabel("Number of blocks")
plt.ylabel("Execution time")

plt.yscale('log')
plt.show()
```



코어 4개를 1번 실행시킨 결과

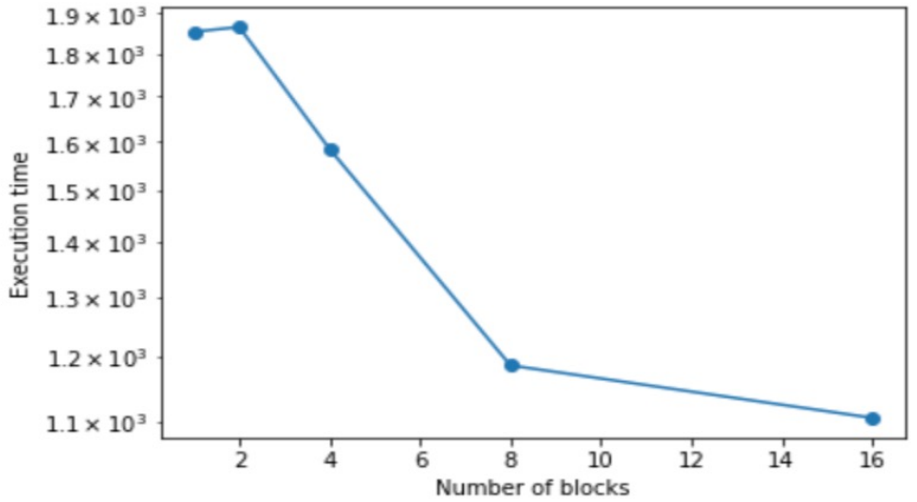
3 Experiment Result

✓ Core 4개인 경우 (제공된 데이터셋)

```
the number of blocks: 16
duration: 1104.9789338111877

the number of blocks: 8
duration: 1185.5057711601257

the number of blocks: 4
duration: 1584.4593710899353

the number of blocks: 2
duration: 1866.1784844398499

the number of blocks: 1
duration: 1854.3250815868378
```

그래프

```
plt.plot(block, execution_time, marker = 'o')

plt.xlabel("Number of blocks")
plt.ylabel("Execution time")

plt.yscale('log')
plt.show()
```



코어 4개를 1번 실행시킨 결과

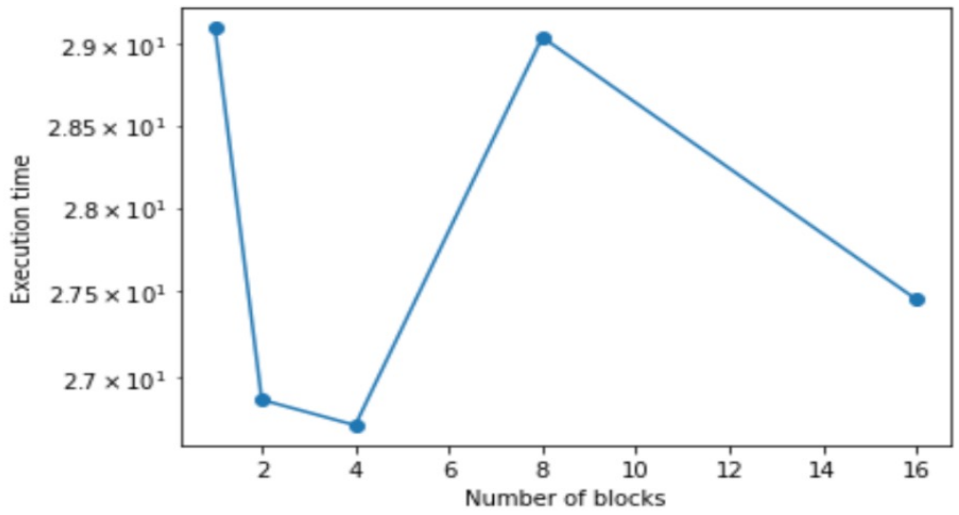
1 Experiment Result

✓ Core 1개인 경우 (직접 만든 데이터셋)

```
the number of blocks: 16
average duration for 5 times: 27.454771709442138

the number of blocks: 8
average duration for 5 times: 29.045183897018433

the number of blocks: 4
average duration for 5 times: 26.722447633743286

the number of blocks: 2
average duration for 5 times: 26.868903875350952

the number of blocks: 1
average duration for 5 times: 29.102856397628784
```

그래프

```
plt.plot(block, execution_time, marker = 'o')

plt.xlabel("Number of blocks")
plt.ylabel("Execution time")

plt.yscale('log')
plt.show()
```



코어 1개를 각각 5번 실행시킨 평균 결과

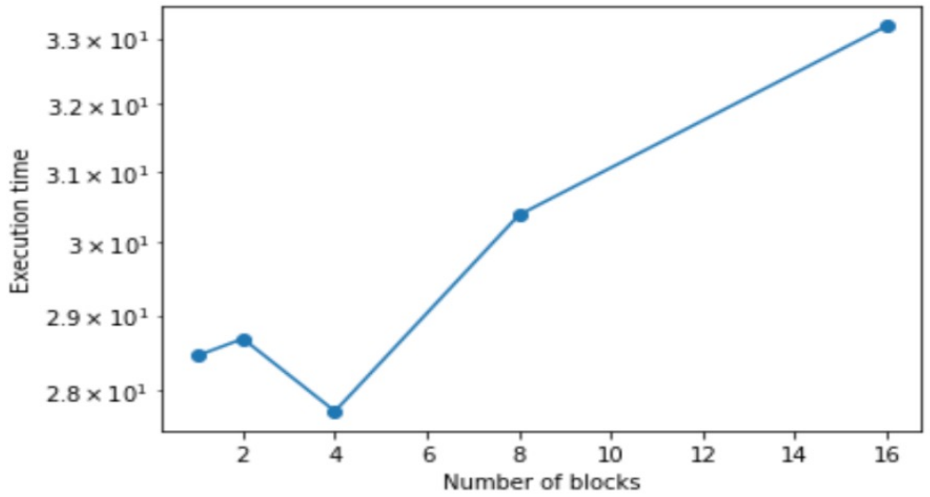
1 Experiment Result

✓ Core 2개인 경우 (직접 만든 데이터셋)

```
the number of blocks: 16
average duration for 5 times: 33.192336082458496

the number of blocks: 8
average duration for 5 times: 30.386149740219118

the number of blocks: 4
average duration for 5 times: 27.72416310310364

the number of blocks: 2
average duration for 5 times: 28.680760145187378

the number of blocks: 1
average duration for 5 times: 28.452122688293457
```

그래프

```
plt.plot(block, execution_time, marker = 'o')

plt.xlabel("Number of blocks")
plt.ylabel("Execution time")

plt.yscale('log')
plt.show()
```



코어 2개를 각각 5번 실행시킨 평균 결과

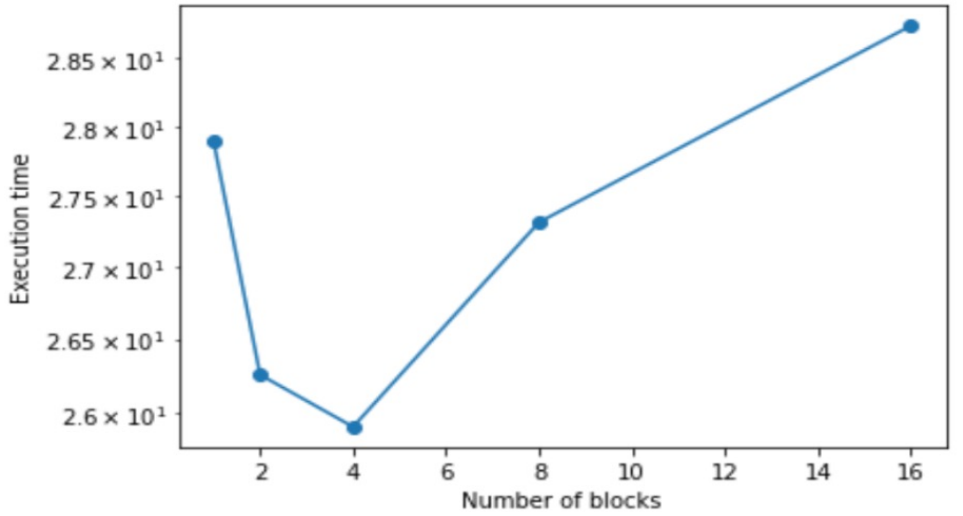
1 Experiment Result

✓ Core 4개인 경우 (직접 만든 데이터셋)

```
the number of blocks: 16
average duration for 5 times: 28.740951251983642

the number of blocks: 8
average duration for 5 times: 27.31389708518982

the number of blocks: 4
average duration for 5 times: 25.903746461868288

the number of blocks: 2
average duration for 5 times: 26.255461931228638

the number of blocks: 1
average duration for 5 times: 27.889938592910767
```

그래프

```
plt.plot(block, execution_time, marker = 'o')

plt.xlabel("Number of blocks")
plt.ylabel("Execution time")

plt.yscale('log')
plt.show()
```



코어 4개를 각각 5번 실행시킨 평균 결과

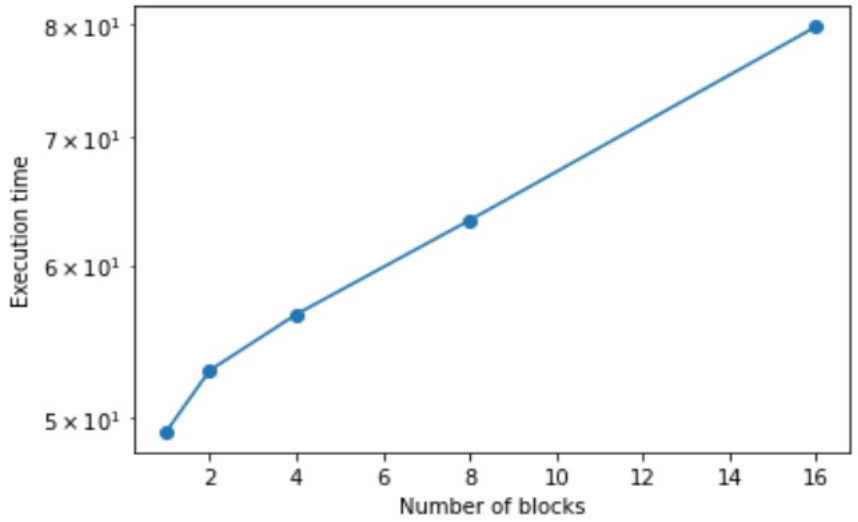
2 Experiment Result

✓ Core 4개인 경우 (직접 만든 데이터셋)

```
the number of blocks: 16
duration: 79.81592845916748

the number of blocks: 8
duration: 63.307430267333984

the number of blocks: 4
duration: 56.5703125

the number of blocks: 2
duration: 52.89344239234924

the number of blocks: 1
duration: 49.15682792663574
```

그래프

```
plt.plot(block, execution_time, marker = 'o')

plt.xlabel("Number of blocks")
plt.ylabel("Execution time")

plt.yscale('log')
plt.show()
```



코어 4개를 1번 실행시킨 결과

Team Conclusion

1번째 시도에서 성능 결과가 안좋게 나와서 관련 자료를 찾아보던 중
GroupByKey가 Shuffle 측면에서 비효율적이라는 것을 알게 되었고

같은 키를 같은 파티션에 넣는 방향으로 PartitionBy를 사용해서 2번째 시도를 했지만
1번째 시도와 마찬가지로 성능이 안좋게 나왔다.

그래서 블록을 만드는 데 필요한 데이터 개수 자체를 줄이는 방향으로 3번째 시도를 했고
Pair 쌍 대신에 Link Structure로 블록을 생성한 결과가 성능이 가장 좋았다.

Team Analysis

1. Partition & Core 관계

하나의 파티션은 하나의 Core가 맡는다.

즉, 파티션의 크기가 결국 Core 당 필요한 메모리 크기를 결정한다.

Partition 수 - Core 수 , Partition 크기 - 메모리 크기

따라서 Partition의 크기와 수가 Spark의 성능에 큰 영향을 미치는데,

각 코어는 동시에 여러 파티션을 맡을 수 없다. 즉, 오히려 파티션이 많으면 더 오래 걸릴 수 있다.

Team Analysis

1. Partition & Core 관계

파티션의 개수가 너무 많으면 오버헤드가 커져 오히려 성능이 악화될 수 있다.

스파크 드라이버가 모든 파티션의 메타데이터를 보관해야 하고

각 파티션을 스케줄링 하는 시간이 들기 때문에 코어 수보다 파티션이 너무 많은 경우도 안좋다.

파티션의 개수가 코어 개수보다 적으면 일부 코어는 작업을 안하고 쉬는 상태가 된다.

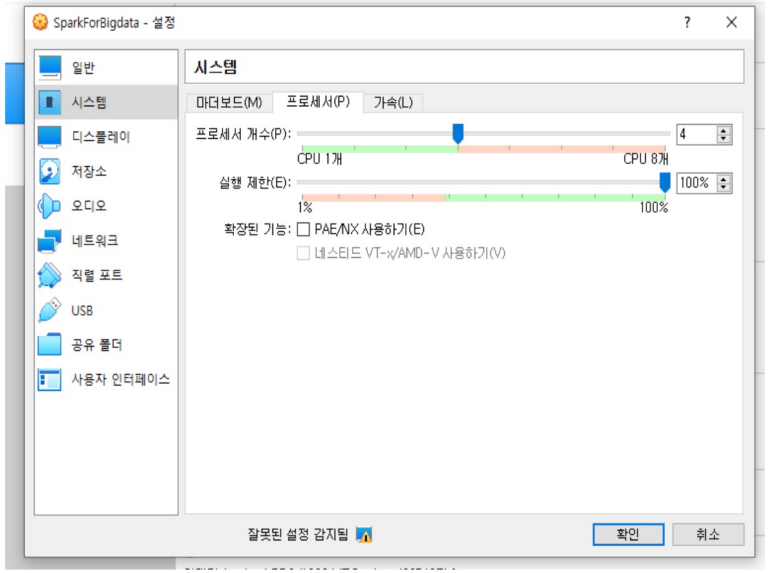
파티션의 개수는 코어 개수보다 크거나 같은 것이 효율적이다.

Team Analysis

2. 서버 코어 & 실행 코어 관계

서버 코어 수보다 많은 executor 코어 수는 할당할 수 없다.

Spark를 이용하여 알고리즘을 구현할 때, 서버 코어 수 확인이 필수다.



VM 의 Default (기본) 코어 수는 1개이기 때문에 바꿔주었다.

Team Analysis

3. 성능 개선

Map, Key by 등으로 새로운 RDD를 생성할 때는 파티션이 변경되지 않는다.

Key-Value로 이루어진 pairRDD의 경우
각각의 파티션들이 key와는 상관없는 데이터들을 랜덤하게 가지게 되고,
이는 key를 기반으로 하는 GroupByKey와 같은 연산을 할 경우 많은 셔플링이 발생하게 된다.

따라서 같은 key를 가진 데이터들을 같은 파티션에 모이도록 하면 성능을 개선할 수 있다.

More Idea ..

[Block Rank] 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 호스트 끼리 연결된 링크

     www.naver.com/22/333 <-> www.hanyang.ac.kr/111/22

2. The GeoCites Effect

   = 친구추천 알고리즘에서 친구가 많은 노드와 같은 효과

   작은 호스트들이 제거되면, 큰 호스트는 높은 intra 호스트 밀도를 갖게 된다.

   이때 매우 적은 호스트들만 GeoCites 효과를 겪는다.

More Idea ..

3. BlockRank

1) 도메인에 따라 웹을 블록으로 나눈다.

2) 각 블록에 대한 로컬 페이지랭크를 계산한다.

3) 각 블록에 대한 블록 랭크, 즉 상대적 중요성을 평가한다.

4) 블록 랭크 값을 이용해서 블록 내의 로컬 페이지랭크에 가중치를 부여하고,
이를 통해 전체 페이지 랭크 벡터를 추정한다.

5) 이 추정된 페이지랭크 벡터를 초기값으로 사용한다.

4. 주요 장점

Caching effect에 대한 성능 향상, 로컬 페이지랭크 벡터는 빨리 수렴한다.

위에서 언급한 단계 2)의 로컬 페이지랭크 계산은 완전히 병렬적으로 이뤄지며,

재활용이 가능하다. 지역 참조를 감소시키고, 계산 복잡도 감소되며 높은 병렬성을 지닌다.

계산 결과 재활용 가능: 개인화된 PageRank를 전체 PageRank 계산에 활용할 수 있다.

Thank you

1조의 발표였습니다. 감사합니다.