KIPS Tr. Comp. and Comm. Sys.
Vol.7, No.1 pp.9~18 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2018.7.1.9

Performance Optimization Strategies for Fully Utilizing Apache Spark

Rohyoung Myung† · Heonchang Yu†† · Sukyong Choi†††

ABSTRACT

Enhancing performance of big data analytics in distributed environment has been issued because most of the big data related applications such as machine learning techniques and streaming services generally utilize distributed computing frameworks. Thus, optimizing performance of those applications at Spark has been actively researched. Since optimizing performance of the applications at distributed environment is challenging because it not only needs optimizing the applications themselves but also requires tuning of the distributed system configuration parameters. Although prior researches made a huge effort to improve execution performance, most of them only focused on one of three performance optimization aspect: application design, system tuning, hardware utilization. Thus, they couldn't handle an orchestration of those aspects. In this paper, we deeply analyze and model the application processing procedure of the Spark. Through the analyzed results, we propose performance optimization schemes for each step of the procedure: inner stage and outer stage. We also propose appropriate partitioning mechanism by analyzing relationship between partitioning parallelism and performance of the applications. We applied those three performance optimization schemes to WordCount, Pagerank, and Kmeans which are basic big data analytics and found nearly 50% performance improvement when all of those schemes are applied.

Keywords : Apache Spark, Performance Optimization, System Tuning

아파치 스파크 활용 극대화를 위한 성능 최적화 기법

명 노 영† · 유 헌 창†† · 최 수 경†††

요 약

분산 처리 플랫폼에서 다양한 빅 데이터 처리 어플리케이션들의 수행 성능 향상에 대한 관심이 높아지고 있다. 이에 따라 범용적인 분산 처리 플랫폼인 아파치 스파크에서 어플리케이션들의 처리 성능 최적화에 대한 연구들이 활발하게 진행되고 있다. 스파크에서 데이터 처리 어플리케이션들의 수행 성능을 향상시키기 위해서는 스파크의 분산처리모델인 Directed Acyclic Graph(DAG)에 알맞은 형태로 어플리케이션을 최적화시켜야 하고 어플리케이션의 처리 특징을 고려하여 스파크 시스템 파라미터들을 설정해야 하기 때문에 매우 어렵다. 기존 연구들은 각각의 어플리케이션의 처리 성능에 영향을 주는 하나의 요소에 대한 부분적인 연구를 수행했고, 최종적으로 어플리케이션의 성능 개선을 이뤄냈지만 스파크의 전반적인 처리과정을 고려한 성능 최적화를 다루지 않았을 뿐만 아니라 처리성능과 상관관계를 갖는 다양한 요소들의 복합적인 상호작용을 고려하지 못했다. 본 연구에서는 스파크에서 일반적인 데이터 처리 어플리케이션의 수행 과정을 분석하고, 분석된 결과를 토대로 어플리케이션의 처리과정 중 스테이지 내부와 스테이지 사이에서 성능 향상을 위한 처리 전략을 제안한다. 또한 스파크의 시스템 설정 파라미터 중 분산 병렬처리와 밀접한 관계를 갖는 파티션 병렬화에 따른 어플리케이션의 수행성능을 분석하고 적합한 파티셔닝 최적화 기법을 제안한다. 3가지 성능 향상 전략의 실효성을 입증하기 위해 일반적인 데이터 처리 어플리케이션: WordCount, Pagerank, Kmeans에 각각의 방법을 사용했을 때의 성능 향상률을 제시한다. 또한 제안한 3가지 성능 최적화 기법들이 함께 적용될 때 복합적인 성능향상 시너지를 내는지를 확인하기 위해 모든 기법들이 적용됐을 때의 성능 향상률을 제시함으로써 본 연구에서 제시하는 전략들의 실효성을 입증한다.

키워드 : 아파치 스파크, 성능 최적화, 시스템 튜닝

1. 서 론

빅데이터의 시대가 도래하면서 다수의 연산자원들이 병렬적으로 작업을 처리하는 분산시스템에 대한 관심이 높아졌다. 이러한 분산시스템에서의 작업들은 다수의 노드들로 구성되는 복합적인 시스템에서 처리되기 때문에 기존의 단일 노드에서 작업을 처리하기 위해 설계된 프로그래밍 모델들

※ 이 연구는 2017년도 미래창조과학부의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(NRF-2016R1C1B1012742).
† 준 회 원 : 고려대학교 컴퓨터학과 박사과정
†† 종신회원 : 고려대학교 컴퓨터학과 교수
††† 비 회 원 : 고려대학교 정보창의교육연구소 연구교수
Manuscript Received : August 7, 2017
Accepted : August 22, 2017
* Corresponding Author : Sukyong Choi(csukyong@korea.ac.kr)

을 분산시스템에 적용하기 어렵다. [1]은 이러한 문제를 해결하기 위해 개발된 분산처리 프로그래밍 모델로 하나의 거대한 작업을 다수의 맵과 리듀스 단계의 작업으로 분할하여 반복적으로 맵 리듀스 단계를 거쳐 작업을 처리하는 방식으로 작업들을 분산시켜서 병렬 처리한다.

그러나 1세대 맵 리듀스 프레임워크인 하둡의 맵 리듀스의 경우 한번의 맵 리듀스 작업 마다 이를 위한 쓰레드를 생성하고 소멸시키기 때문에 반복적인 맵 리듀스 작업에 대한 비용이 많이 들고, 사용자가 직접 작업을 맵 리듀스 형태로 개발해야 하는 등 시스템 활용에 대한 많은 어려움이 있었다. 또한 리듀스 과정 중 데이터 셔플링으로 인한 성능 저하가 발생하는 문제점을 갖고 있다. 2세대 분산 처리 시스템들[2, 3]은 사용자가 보다 쉽게 분산 처리 시스템에 알맞은 형태로 작업을 개발할 수 있도록 다양한 라이브러리를 지원할 뿐만 아니라 반복적인 맵 리듀스 작업을 효율적으로 처리할 수 있도록 재사용되는 데이터를 메모리에 지속적으로 놓고 쓰는 인 메모리 컴퓨팅을 적용해서 성능을 향상시켰다.

스파크의 경우 DAG를 기반으로 작업들을 분산해서 처리하고, 스트리밍 데이터, 데이터베이스, RAW 데이터 등 다양한 종류의 데이터를 처리하기 위한 [4, 5] 등의 내장 라이브러리를 제공한다. 또한 스파크에서는 분산 처리 시스템상에서 작업 처리 도중 발생할 수 있는 다양한 경우의 오류를 극복할 수 있도록 RDD를 고안하고 RDD들의 처리과정인 lineage를 작업이 완료될 때까지 관리한다. 이를 통해 작업 처리 도중 오류가 발생하더라도 문제가 발생한 RDD를 처리하는 단계부터 이어서 작업을 처리할 수 있기 때문에 많은 분산 처리 시스템 사용자들에게 각광받고 있다.

이러한 시스템들은 작업 처리를 관리하는 마스터 노드와 분산된 작업들을 처리하는 슬레이브 노드로 구성되며 노드간 통신을 위한 라이브러리, 멀티코어와 같은 시스템 자원을 효율적으로 사용하기 위한 라이브러리 등 다수의 시스템 구성요소들이 복합적으로 상호작용한다. 시스템 내부에서는 작업 처리를 위한 시스템 자원 사용량 설정, 작업의 병렬화 수준 설정 등의 다양한 시스템 설정을 제공하기 때문에 작업의 특징을 고려한 시스템 설정이 잘 이뤄질 경우 처리 성능을 크게 향상시킬 수 있다. 또한 시스템의 처리 과정을 고려하여 작업 자체에 대한 최적화를 할 경우에도 작업 처리에 대한 성능 개선이 이뤄질 수 있다. 분산 처리 시스템은 이러한 최적화 기법의 적용 유무에 따라 작업 처리속도 차이가 극심하게 발생하기 때문에 작업 유형과 처리 환경을 고려한 성능 최적화는 필수적이다. 그러나 시스템의 분산처리 과정이 복잡하기 때문에 처리과정에 대한 전반적인 이해가 어려울 뿐만 아니라 다양한 시스템 설정 옵션들을 처리해야 할 작업의 유형을 고려하여 성능을 최적화해야 하기 때문에 분산 처리 시스템에서 성능 최적화는 매우 어려운 일이다.

본 논문에서는 스파크의 처리 과정과 처리 성능에 영향력이 큰 시스템 설정 옵션을 심도있게 분석하여 다각도에서 처리성능 향상에 기여할 수 있는 최적화 기법들을 연구하고 대표적인 빅 데이터 분석 앱들에 적용하여 해당 기법들에 대한 실효성을 입증한다.

우선 스파크의 구성요소와 구조를 분석하고, 스파크에서 일반적인 데이터 처리 작업이 어떻게 진행되는지에 대해 스테이지 내부, 스테이지 외부 처리로 나눠서 분석한다. 분석 결과를 토대로 단순히 하나의 구성요소에 대한 최적화가 아닌 작업이 처리되는 스테이지 내부 처리단계, 스테이지 외부 처리단계, 시스템 파라미터 설정 단계에 알맞은 최적화 전략을 모색하고 적용한다.

첫 번째 스테이지 내부처리 최적화를 위해 트랜스포메이션들이 일반적으로 데이터에 대한 반복적인 연산 작업을 통해 새로운 RDD를 생성하는 사실을 바탕으로 스테이지 내부 처리에 대한 연산 비용을 수식으로 모델링한다. 수식화된 모델에서 반복 작업에 대한 처리 시간을 최소화하는 파티션 폴딩 스킴을 제안한다. 그리고 폴딩 스킴을 보편적인 빅 데이터 분석 앱인 WordCount에 데이터 사이즈를 바꿔가면서 적용하여 파티션 폴딩 정도에 따른 작업 처리시간의 변화를 측정하고 작업 처리를 최소화시키는 최적화된 파티션 폴딩 수준을 제시한다. 두 번째로 스테이지 외부처리 과정을 최적화하기 위해 해당 과정에 성능 영향을 미치는 요소인 파티션들의 셔플링 및 소팅에 대해 분석하고, 해당 단계에서의 작업 처리 성능 최적화를 위한 이상적인 엘리먼트의 분포를 제안 및 검증한다. 마지막 단계인 시스템 파라미터 설정에서는 시스템에서의 작업 병렬처리의 최적화를 위해 시스템이 제출된 작업을 분할하고 처리하는 과정과 작업 분할과 연관된 파라미터들을 분석하여 작업 처리 성능 최적화를 위한 작업 분할 전략을 제시한다.

본 연구에서는 각각의 최적화 기법들을 보편적인 데이터 처리 어플리케이션들: WordCount, Pagerank, Kmeans에 적용하여 작업 처리 성능과 관련된 요소들에 미치는 영향을 평가하고, 제안하는 최적화 기법들의 다양한 조합들도 해당 어플리케이션들에 적용하여 각 어플리케이션마다 처리성능을 가장 최적화 할 수 있는 최적화 기법 조합을 제시한다.

본 논문이 기여하는 바는 다음과 같다.

- 스파크의 구조 및 구성요소에 대한 분석을 기반으로 일반적인 스파크 작업이 스파크에서 처리되는 과정을 단계별로 분석한다.
- 스파크의 처리과정 각 단계의 처리성능 최적화를 위한 3가지 최적화 기법: 스테이지 내부/간 처리최적화, 파티셔닝 최적화를 제안한다.
- 가지의 보편적인 빅데이터 분석 기법: WordCount, Pagerank, Kmeans에 대해 각각의 최적화 기법들과 최적화 기법들의 조합을 적용하여 처리 시간과 작업 할당 편차를 감소시킨다.

이후 2장 연구배경에서는 본 연구의 배경이 되는 개념들을 다룬다. 3장 작업 처리 과정 분석에서는 아파치 스파크의 구조 및 작업이 처리되는 과정을 다룬다. 4장 성능 최적화에서는 작업 처리 과정을 3단계로 나눠 각 단계별 성능최적화 기법을 제안한다. 5장 실험 결과에서는 각각의 성능 최적화 기법 및 기법들의 조합에 대한 성능향상 결과를 제시한다. 6장 관련연구에서는 분산처리 시스템에서 성능 최

적화를 위한 선행연구들을 소개한다. 마지막으로 7장 결론에서는 연구결과를 요약하고 추후 연구를 소개한다.

2. 연구 배경

2.1 아파치 스파크

아파치 스파크[1]은 결함포용, 인 메모리 컴퓨팅, DAG 기반 분산처리를 지원하는 범용 분산 처리 시스템이다. 스파크는 작업 처리를 관리하는 드라이버(마스터 노드에 위치)와 드라이버로부터 처리해야 할 분산된 작업들을 할당 받아 처리하고 결과를 반환하는 하나 이상의 엑세큐터(Executor, 슬레이브 노드에 위치)들로 구성된다. 스파크는 클러스터 시스템의 자원 관리를 위해 자원 관리자인 standalone cluster manager를 기본적으로 제공하고 하둡(Hadoop) 얀(Yarn), 메소스(Mesos)와 같은 외부 클러스터 자원관리자도 지원한다.

아파치 스파크는 3가지 방법을 통한 시스템 옵션 설정을 지원한다. 첫 번째로 어플리케이션 내부에서 sparkconf 파일을 설정하여 시스템 옵션을 설정할 수 있다. 두 번째로 파이썬 쉘, 스칼라 쉘과 같은 인터랙티브 쉘에서 sparkcontext를 초기화할 때 시스템 파라미터를 설정하는 방식을 통해 시스템 옵션을 수정할 수 있다. 마지막으로 드라이버를 실행하기 전에 deaults.conf 파일 변경을 통해 추가적인 설정이 없을 때의 파라미터들의 기본 값을 수정할 수 있다.

클러스터 자원 매니저로 standalone을 사용할 경우 드라이버 및 엑세큐터가 사용할 CPU의 코어 개수와 메모리 크기, 엑세큐터 내에서 처리될 작업 하나가 사용할 CPU 코어의 개수, 병렬처리 수준, 파티션의 최대 크기 등 시스템 자원 및 처리할 작업과 상관 관계를 갖는 옵션들을 설정할 수 있다. 이러한 시스템 옵션들의 설정 최적화 여부에 따라 작업 처리 성능차이가 크게 발생한다[6].

2.2 DAG기반 분산처리

맵 리듀스[1]은 분산 처리 시스템에서 대규모 작업을 보다 쉽게 병렬화하여 처리할 수 있도록 만들어진 프로그래밍 모델로서 일반적인 빅 데이터 분석기법들뿐만 아니라 딥러닝과 같은 최신 기계학습 기법도 분산 병렬화시켜서 처리할 수 있다[7, 8].

이에 반해 현재 분산 처리 프레임워크들[2, 9]는 DAG를 기반으로 작업들을 분산시켜서 처리한다. 이러한 DAG 기반 분산 처리는 맵 리듀스에 비해 다양한 작업들을 처리할 수 있고 프레임워크 자체에서 인메모리 컴퓨팅을 지원하기 때문에 반복적인 작업들에 대해서도 성능저하가 크게 발생하지 않는다. 아파치 스파크의 경우 DAG 기반 분산 작업은 스테이지를 기준으로 수행되고 스테이지 내부 처리와 스테이지간 처리로 나눌 수 있다. 스테이지 내부처리에서는 narrow dependency 유형을 갖는 일련의 작업들을 처리하고, 스테이지간 처리에서는 wide dependency 유형을 갖는 작업을 수행하는 구간으로 파티션간 혹은 RDD간 셔플링이 발생하기 때문에 입력 값으로 사용되는 데이터에 대한 스테이지 내부 작업들이 완료된 후에 셔플링 작업을 수행한다. 각 과정에 대한 설명과 분석은 3장에서 다룬다.

3. 아파치 스파크의 구조 및 작업 처리 과정 분석

3.1 아파치 스파크의 구조

스파크 클러스터는 Fig. 1과 같이 분산 클러스터 자원 및 작업 처리를 관리하는 마스터 노드와 실제 분산된 작업을 처리하는 1개 이상의 슬레이브 노드로 구성된다. 마스터노드는 크게 클러스터 자원을 관리하는 Standalone Cluster Manager와 작업 처리를 관리하는 스파크 드라이버로 구성된다.

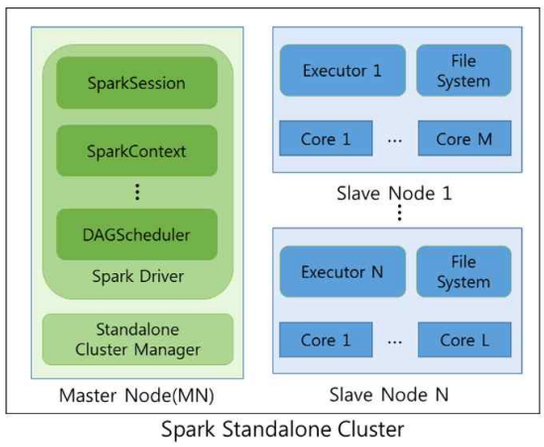

Fig. 1. Spark Cluster Architecture

스파크 드라이버는 RDD에 대한 생성 및 작업을 수행하는 다양한 API를 제공하는 SparkSession 및 SparkContext와 작업처리과정 생성 및 작업 처리를 감독하는 DAGScheduler 등으로 구성된다. 또한 스파크 드라이버는 사용자로부터 각 엑세큐터가 사용할 코어의 개수와 메모리 및 분산된 하나의 단일 작업이 사용할 CPU의 개수를 입력받아서 유연하게 스파크 클러스터의 자원을 활용한다.

슬레이브 노드는 드라이버로부터 전달받은 작업을 처리할 엑세큐터를 생성하고 관리한다. 실제로 분산된 작업은 슬레이브 노드에서 생성된 엑세큐터에서 처리된다. 각각의 엑세큐터는 스파크 드라이버로부터 전체 작업 중 일부분을 전달받아서 자신이 사용 가능한 모든 코어를 사용하여 작업을 병렬로 처리한다.

3.2 작업 처리 과정

스파크 드라이버로 제출된 작업은 Fig. 2와 같이 1개 이상의 스테이지를 거쳐서 처리된다. 스파크에서는 RDD라는 추상 자료형태를 통해 데이터를 표현한다. RDD는 1개 이상의 파티션으로 구성되며, 각각의 파티션은 1개 이상의 엘리먼트를 갖는다. RDD는 반복처리가 가능한 자료구조를 갖기 때문에 엘리먼트는 반복처리가 되는 최소 단위를 의미한다. 스파크는 작업을 병렬로 처리하기 위해 RDD를 파티션단위

로 나눈 후 스파크 스케줄링 정책에 따라 각각의 엑세큐터에게 파티션집합에 대한 처리를 지시하여 파티션들에 대한 작업(task) 집합을 처리한다. 스파크 드라이버가 RDD를 분할하는 방법과 분할된 RDD를 엑세큐터에서 병렬로 처리하는 세부 과정은 4.3절에서 별도로 설명한다.

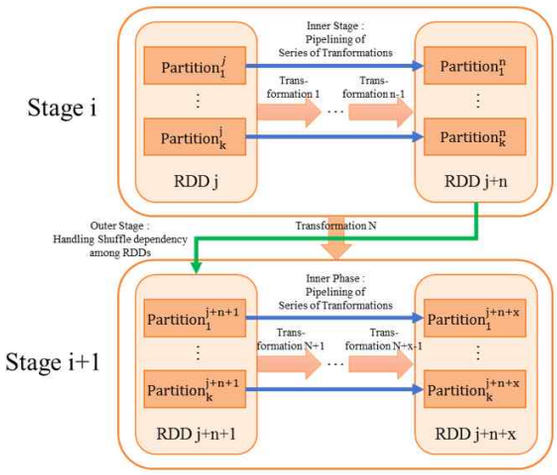

Fig. 2. Job Executing Procedure

RDD에 대한 작업은 wide dependency와 narrow dependency 유형을 갖는 트랜스포메이션과 실제작업처리를 시작하기 위한 엑션[10]으로 나뉘며 스파크는 lazy execution을 사용하기 때문에 액션이 호출돼야 이전까지 사용했던 트랜스포메이션들에 대한 처리가 수행된다. 하나의 스테이지에서는 wide dependency 유형의 트랜스포메이션이 호출되기 전까지 일련의 narrow dependency 유형의 트랜스포메이션들을 RDD의 엘리먼트에 반복적으로 적용하여 새로운 RDD를 생성한다. 이후에 wide dependency 트랜스포메이션이 호출되면 해당 트랜스포메이션에서 사용되는 RDD들의 파티션들에 대한 셔플링 작업 및 키 값을 기반으로 한 작업을 처리하여 새로운 RDD를 생성한다. 이 작업이 완료되면 다음 스테이지에서 다시 생성된 RDD에 일련의 narrow dependency 유형의 트랜스포메이션들을 적용하여 새로운 RDD들을 생성하는 방식으로 제출된 작업을 처리한다. 결과적으로 스파크에서 처리되는 작업은 1개 이상의 스테이지에서 narrow dependency 유형의 일련의 트랜스포메이션들을 처리하는 스테이지 내부 처리 단계와 스테이지 사이에서 wide dependency 유형의 작업을 처리하는 스테이지간 처리 단계로 나눌 수 있다.

4. 성능 최적화

스파크와 같은 분산 처리 시스템들은 시스템을 구성하는 요소들의 성능, 처리할 작업의 특징, 처리해야 할 작업의 양, 시스템의 설정 등 복합적인 요인에 의해 성능에 영향을 받는다. 본 연구에서는 위와 같은 요소들을 고려해서 작업 처리 과정 및 처리할 작업의 특징을 고려하여 스테이지 내부 처리 단계에서와 스테이지간 처리 단계에서의 성능최적화, 그리고 시스템을 구성하는 요소들의 성능 및 이를 활용하기 위한 시스템 설정을 고려하여 시스템 병렬화 설정 최적화를 통한 통합적인 작업 처리 시간 최적화 기법을 제안한다.

4.1 스테이지간 내부 처리 최적화

이 절에서는 스테이지 내부 처리 단계에 대한 모델을 만들고, 이를 통해 실제 시스템에서 동작하는 어플리케이션들에 파티션 폴딩 스킴을 적용하여 결과 값에 대한 분석을 통해 제안된 최적화 기법의 실효성을 입증한다. 해당 과정에서 엑세큐터는 스파크 드라이버로부터 할당받은 작업 집합에 해당되는 파티션들의 엘리먼트들을 순회하며 입력데이터에 대해 일련의 트랜스포메이션을 각각의 엘리먼트에 적용하여 새로운 RDD를 생성한다. 이때 하나의 스테이지에서 하나의 파티션에 일련의 트랜스포메이션을 적용하는 것을 작업이라고 정의한다. 우리는 Equation (1)을 통해 하나의 스테이지에서 하나의 파티션을 처리하는 과정을 모델링한다. Equation (1)에서 사용되는 노테이션들은 Table 1과 같다.

Table 1. Notations for Modeling Inner Stage Processing Optimization

α	Static execution time except for iterative transformation set
e_n	The number of elements in single partition
e_s	The size of elements
r_s	The average size of records
r_n	The number of records in single partition
$f(x)$	function that measure the time of processing single transformation in a stage

$$Inner\,Stage\,Cost = e_n(\alpha + f(e_s)) \quad (1)$$

$$(e_s = (r_s + r_n)) \quad (2)$$

$$Inner\,Stage\,Cost = e_n(\alpha + f(r_s + r_n)) \quad (3)$$

하나의 파티션에 포함된 엘리먼트들의 개수하나의 스테이지에서 단일 작업을 처리하는데 걸리는 시간은 Equation (1)과 같이 한 파티션의 엘리먼트 개수와 각 라인을 반복적으로 처리하면서 발생하는 부가적인 지연시간과의 곱과 총 엘리먼트 개수와 하나의 트랜스포메이션을 처리하는 데 필요한 시간과 곱의 합으로 나타낼 수 있다. 트랜스포메이션을 처리하는데 필요한 시간은 엘리먼트의 크기와 상관관계를 갖기 때문에 입력파라미터로 엘리먼트의 크기를 사용한다. 엘리먼트의 크기는 Equation (2)에 따라 엘리먼트를 구성하는 최소단위의 레코드 크기와 레코드의 개수의 곱으로 나타낼 수 있다. 따라서 하나의 트랜스포메이션에서 작업을 처리하는데 걸리는 비용은 Equation (3)과 같다.

본 연구에서 제안하는 바는 파티션 내부의 엘리먼트들을 반복처리하는데 발생하는 비용을 감소시키는 파티션 폴딩 스킴이다. 파티션 폴딩 스킴은 한 엘리먼트에 포함되는 레코드들을 k배 만큼 증가시켜서 전체 엘리먼트의 수를 1/k로 감소시키는 기법이다. 파티션 폴딩 스킴을 적용할 경우 작업 처리 시간은 Equation (4)와 같다.

$$Inner\,Stage\,Cost = \frac{1}{k}e_n(\alpha + f(k^*e_s)) \quad (4)$$

$$partition\,size = e_n{}^*e_s = k^*e_n{}^*\frac{1}{k}{}^*e_s \quad (5)$$

$$Inner\,Stage\,Cost = \frac{1}{k}e_n(\alpha + f(k^*r_n{}^*r_s)) \quad (6)$$

k만큼 파티션을 폴딩 시킬 경우 전체 엘리먼트들의 수는 1/k로 감소하고 반복문에 대한 고정처리시간은 상수기 때문에 그대로 유지된다. 그리고 엘리먼트의 크기가 k배로 증가하면서 트랜스포메이션 처리 시간은 Equation (4)가 된다. 파티션에 트랜스포메이션을 적용하는데 걸리는 시간은 Equation (5)에 따라 Equation (6)과 같이 나타낼 수 있다. 결과적으로 정적 처리시간, 처리시간 함수 둘 다 1/k로 감소된다.

본 논문에서는 엘리먼트의 개수가 감소하면서 감소되는 정적 처리시간과 엘리먼트의 크기가 증가하면서 증가하는 처리시간 함수의 반비례 관계를 알아보기 위해 파티션 폴딩 수준을 증가시키면서 1GB의 데이터에 WordCount를 실행해 보았다. 실험 결과는 Fig. 3과 같다.

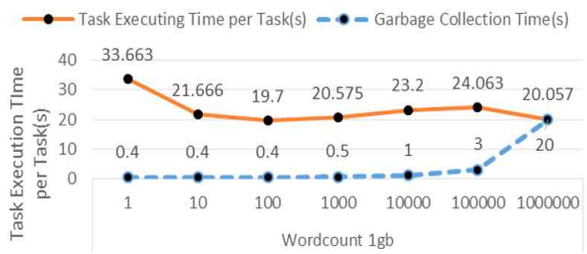

Fig. 3. Execution Time of WordCount with Inner Stage Optimization

엘리먼트의 개수를 줄이면서 엘리먼트당 레코드들의 개수를 늘리면 데이터의 각 엘리먼트를 조회하면서 부가되는 고정 처리시간과 처리 시간 함수 값은 조회해야 할 엘리먼트의 개수에 비례해서 줄어든다. 그러나 트랜스포메이션의 순수 처리 시간 함수는 엘리먼트당 레코드 개수를 인자로 갖기 때문에 줄어드는 엘리먼트 개수의 비율만큼 늘어난 엘리먼트 당 레코드 수는 함수의 결과 값을 증가시키게 된다. 이 반비례 관계는 스파크가 구동되는 환경(워커의 개수 및 연산능력, 엑세큐터의 개수 및 연산 능력 등)과 트랜스포메이션 종류에 따라 다르지만 5장에서 밝히듯이 대체적으로 엘리먼트당 레코드 개수가 100일 때 가장 좋은 효율을 보였다. 이후에는 다시 작업 처리시간이 증가하다가 폴딩 수준이 1,000,000이 되면 다시 감소한다. 그러나 파티션 폴딩 수준이 1,000,000을 넘어가면 엘리먼트 크기가 커지기 때문에 Fig. 3과 같이 가비지 콜렉션에 대한 시간이 급상승한다.

4.2 스테이지간 처리 최적화

스테이지간 처리 단계에서는 RDD에 wide dependency 유형의 트랜스포메이션을 적용해서 새로운 RDD를 생성한다. Wide dependency 유형의 트랜스포메이션들은 슬레이브 노드간 데이터 전송을 발생시키는 셔플링을 발생시킨다. 셔플링 단계에서 처리되는 작업들은 일반적으로 엘리먼트들의 키 값을 기준으로 처리되기 때문에 RDD에서 키 값에 대한 엘리먼트 분포에 따라 성능차이가 크게 발생한다. 스파크 2 버전이상의 경우 디폴트 셔플링 정책으로 tungsten sort를 사용하고 필요할 경우 사용자가 자신의 작업에 알맞은 형태의 셔플링 정책을 개발하여 사용할 수도 있다.

RDD내부에서 어떤 키값을 갖는 엘리먼트들을 어디에 위치시킬지에 대한 문제는 키값에 대한 데이터의 분포를 설정하는 문제로 연결된다. 셔플링 과정에서 지연시간을 감소 시키기 위해서는 작업 처리에 알맞은 형태로 엘리먼트들을 분포시키는 것이 중요하다. 본 연구에서는 한 파티션내에 최대한 동일한 키 값을 갖는 엘리먼트들을 위치시키는 경우(분포가 편향된 상태)와 한 파티션내에 모든 키 값에 대해 엘리먼트들을 균등하게 분포시키는 경우(분포가 균등한 상태)에 대해 WordCount와 Pagerank의 처리성능을 측정했다. 두 가지 앱 모두 다 데이터를 편향된 상태로 만들어서 사용할 경우 셔플링 단계에서 소팅에 걸리는 시간과 부가적인 지연시간을 감소시켜서 처리성능을 향상시키는 것을 확인할 수 있었다.

4.3 파티션 병렬 최적화

스파크는 RDD를 생성할 때 기본적으로 사용가능한 모든 코어들의 개수로 RDD를 파티셔닝한다. 파티션 병렬 최적화에서 사용되는 기호들은 Table 2에서 설명한다. 만약 입력 데이터의 크기가 Equation (7)이 될 경우 Equation (8)에 따라 RDD가 파티셔닝되고 작업 처리시 스파크 드라이버의 스케줄링 정책에 따라 파티션들을 가용한 엑세큐터에 할당한다.

Table 2. Notations for Modeling Parallel Optimization

d_s	The size of input data
c_n	The number of available cores
p_s	The size of single partition
p_n	Total number of partitions

$$d_s > c_n{}^*p_s \quad (7)$$

$$p_n = \lceil d_s{}^*p_s \rceil \quad (8)$$

$$p_n = \beta^*c_n \quad (9)$$

스파크의 기존 스케줄링 정책은 fair 스케줄링으로 최대한 모든 엑세큐터가 고르게 작업을 처리하도록 작업집합을

할당한다. 따라서 파티셔닝 수준이 잘못 설정될 경우 엑세큐터간 작업을 처리하는 시간의 편차가 증가하고 전체 작업을 처리하는 시간도 증가하게 된다. 반대로 파티셔닝 수준이 입력 데이터와 처리 환경에 적합하게 설정될 경우 모든 엑세큐터가 쉬는 시간 없이 균등하게 작업을 처리하기 때문에 전체 작업을 처리하는 시간도 감소하게 된다.

스파크 2버전에서는 sparkcontext와 sparksession에 있는 API를 사용해서 파일로부터 RDD를 생성할 수 있다. Sparksession의 경우 RDD를 사용가능한 모든 코어들의 개수로 파티셔닝하고 입력데이터가 Equation (7)인 경우 추가 데이터들도 128MB 단위로 나눠서 Equation (8)과 같이 파티셔닝한다. 이와 다르게 sparkcontext는 사용자로부터 파티셔닝 수준을 입력받아서 입력데이터를 파티셔닝할 수 있다. 본 연구에서는 Equation (7)의 경우에 대해 파티셔닝 수준을 Equation (9)에 따라 설정함으로써 각 엑세큐터들의 처리시간 표준편차를 최소화시키고 결과적으로 작업 처리시간을 감소시킨다.

전체 코어 개수에 곱해지는 값은 3가지 유형의 어플리케이션에서 반복적으로 수행한 실험을 통해 결정된다. RDD에서 새로운 RDD를 만드는 경우에는 리파티셔닝을 통해 파티셔닝 수준을 변경할 수 있다. 그러나 이 경우 리파티셔닝 자체가 wide dependency 유형의 트랜스포메이션이기 때문에 셔플링에 따른 부가적인 지연시간이 발생한다. 그러므로 트랜스포메이션을 적용할 때 리파티셔닝을 할지 여부는 리파티셔닝 자체의 지연시간과 리파티셔닝을 통해 추후에 발생하는 성능 향상간 상관관계를 가늠해서 선택해야 한다.

5. 실험 결과

스파크는 엑션을 호출해야 실제 작업처리가 시작되기 때문에 트랜스포메이션 자체의 성능을 평가하기 위해서는 엑션을 필수적으로 사용해야 하며 해당 엑션은 자체의 처리시간이 적고, 트랜스포메이션의 완전한 처리를 보장해야 한다. take와 같은 엑션은 RDD에 대한 일부만 처리하기 때문에 이전에 수행된 트랜스포메이션의 비용을 측정할 수 없고, saveastextfile은 결과가 disk에 쓰여지는 시간이 추가되기 때문에 atomic한 트랜스포메이션의 비용을 측정하기 어렵다. 본 논문에서는 자체 지연시간이 적고 모든 트랜스포메이션들의 처리를 보장하는 엑션으로 count를 사용했고, 앱 수행 결과가 스파크 드라이버 메모리 크기를 초과하지 않고, 앱 전체에 대한 성능을 측정할 때는 collect를 사용했다.

실험에서는 한대의 마스터 노드와 3대의 슬레이브 노드로 구성된 스파크 클러스터를 사용했고, 각각의 노드들은 운영체제 Ubuntu 14.04를 사용했으며 운영체제를 제외한 부가적인 프로세스들을 종료함으로써 스파크 외 프로그램의 자원 사용을 억제시켰다. 마스터 노드와 슬레이브 노드들은 인텔 코어 i7-6700(3.40GHz*8코어), 16GB 메모리, 256GB SSD, 1GBps 네트워크 대역폭을 사용했다. 그리고 각각의 엑세큐터는 8개의 코어, 10GB 메모리를 사용했다. 모든 실험들은 5회씩 수행 후 평균 값을 최종 결과로 사용했다. 또한 모든 실험들은 스파크의 순수한 성능 측정을 위해 하둡 파일 시스템을 사용하지 않고 각각의 워크로드에서 사용될 모든 데이터의 복사본들이 모든 슬레이브 노드에 있는 환경에서 이뤄졌다. 동일한 맥락에서 스파크 클러스터 관리자도 기본 클러스터관리자인 stand alone manager를 사용했다. 모든 실험결과에서 스테이지 내부 처리 최적화는 Inner stage의 “I”로 표현했고, 스테이지간 처리 최적화는 Outer stage의 “O”, 파티셔닝 최적화는 “P”로 표현했다.

5.1 워크로드

우리는 각각의 최적화 기법과 최적화 기법들의 조합들에 대한 성능을 평가하기 위해 WordCount, Pagerank, Kmeans에 해당 기법들을 적용했다. WordCount는 입력 데이터 내에 있는 각 단어들의 출현 횟수를 계산하는 프로그램이다. 입력데이터는 단어와 단어를 구분하기 위한 식별자로 구성된다. 스파크에서 WordCount를 실행할 경우 엑세큐터는 데이터 파일의 위치를 확인하여 파일이 있을 경우 읽어와서 RDD로 만든다. 이때 파티셔닝 기법에 따라 RDD를 구성하는 파티션의 개수가 정해지고, 각 파티션의 엘리먼트는 일정한 개수의 단어 모음이 된다. 이후 각각의 엑세큐터는 스파크 드라이버로부터 할당받은 파티션들에 대해 일련의 트랜스포메이션을 적용하여 엘리먼트가 (키,값) 순서쌍이 되는 RDD를 생성한다. 이후 셔플링 단계를 거쳐 각각의 키 값에 대한 값들을 합산해서 단어 별 출현횟수를 반환한다.

Pagerank는 입력 데이터 내에 있는 url들이 참조된 횟수를 계산하여 url들의 참조 순위를 계산한다. 입력 데이터는 참조되는 url, 참조하는 url 순서쌍과 이를 구분하기 위한 식별자로 구성된다. 스파크에서 Pagerank를 실행할 때 WordCount로 마찬가지로 엑세큐터가 데이터 파일의 위치를 확인하여 파일이 있을 경우 읽어와서 RDD로 만들어서 스파크 드라이버로부터 할당된 파티션들에 대해 작업을 처리한다. 최종적으로 반환되는 결과물은 모든 url에 대한 (url, url의 중요도) 순서쌍 집합이 된다.

Kmeans는 입력 데이터 내에 있는 n차원 좌표들을 k개의 군집으로 분류한다. 입력 데이터는 n차원 벡터들과 이를 구분하기 위한 식별자로 구성된다. Kmeans를 실행하면 각각의 엑세큐터들은 WordCount 및 Pagerank와 같은 과정을 거쳐서 자신에게 할당된 파티션들에 대한 작업을 처리한다. Kmeans는 k개의 군집의 중심점인 센트로이드들을 입력데이터에서 샘플링한다. 그리고 모든 좌표들에 대해 가장 가까운 센트로이드를 찾아서 군집화한 후 센트로이드들을 군집들의 중심점으로 수정한다. 이때 기존 센트로이드와 새롭게 계산된 센트로이드의 직선거리 변화가 사용자가 설정한 임계값 이하가 될 때까지 반복적으로 센트로이드들을 갱신한다.

5.2 스테이지 내부 최적화 성능

Fig. 4는 WordCount, Pagerank, Kmeans로 1GB, 5GB의 데이터를 처리했을 때 파티션 폴딩 수준을 100의 거듭제곱으로 증가시킬 경우 발생하는 전체 작업 처리 시간 비율이다. WordCount, Pagerank의 경우 1GB, 5GB 데이터 모두에

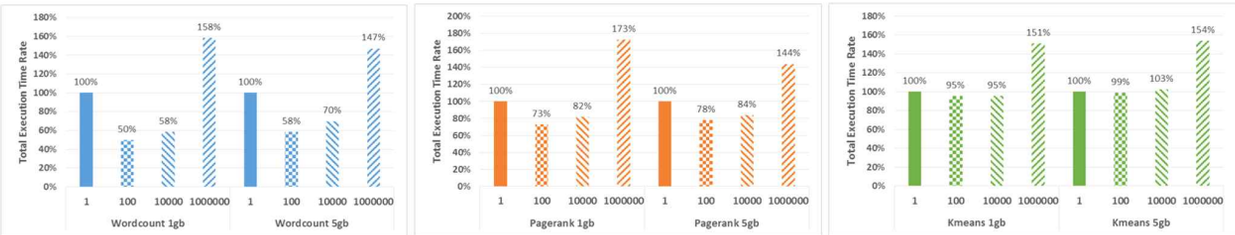

Fig. 4. End-to-end Execution Performance of the Three Applications When "I" Optimization is applied with 1GB and 5GB Data

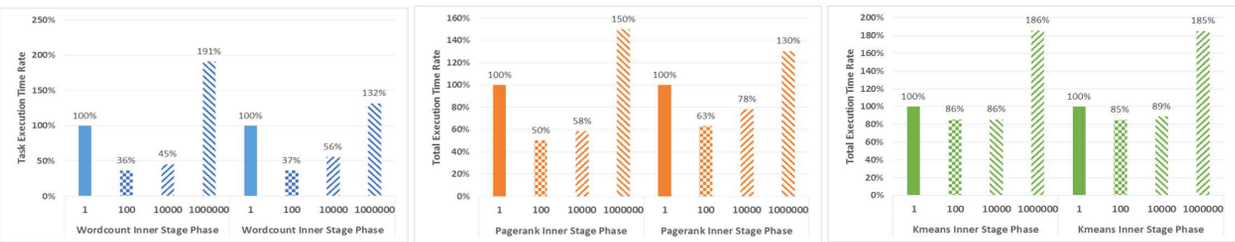

Fig. 5. Inner Stage Execution Performance of the Three Applications When "I" Optimization is applied with 1GB and 5GB Data

서 폴딩 수준이 100일 때 기존 수행시간 대비 수행시간이 각각 50%. 58%, 72%, 78%로 감소하여 가장 성능이 좋았다. Kmeans의 경우 1GB 데이터에 대해서는 폴딩 수준이 10,000일 때 기존 수행시간 대비 수행시간이 95%로 가장 성능이 좋았고, 5GB 데이터에 대해서는 폴딩 수준이 100일 때 기존 수행시간 대비 수행시간이 99%로 가장 좋았다. 그러나 모든 어플리케이션들에 대해 폴딩 수준이 1,000,000이 되면 가비지 콜렉션으로 인한 지연시간이 크게 발생해서 기존보다 성능이 크게 안 좋아졌다.

실제로 파티션 폴딩 기법이 스테이지 내부에서 수행되는 작업들의 성능을 개선하는지 알아보기 위해 전체 작업 중 스테이지 내부 처리에만 소요된 시간의 감소비율을 Fig. 5에 나타냈다. WordCount, Pagerank의 경우 1GB, 5GB 데이터 모두에서 폴딩 수준이 100일 때 기존 수행시간 대비 수행시간이 36%, 37%, 50%, 63%로 가장 성능이 좋아졌다. 또한 Kmeans의 경우도 1GB, 5GB 데이터 모두에서 폴딩 수준이 100일 때 86%, 85%로 성능이 가장 좋았다. Kmeans의 경우 새로운 센트로이드를 계산할 때 반복적인 스테이지간 처리 작업에 의한 지연시간이 크게 발생하기 때문에 전체 처리 시간에 대한 성능 향상보다 스테이지 내부 처리 단계에서의 성능 향상 정도가 더 크게 나타난다.

5.3 스테이지간 최적화 성능

Fig. 6에서는 기존 WordCount, Pagerank와 스테이지간 처리 단계 최적화 적용된 WordCount, Pagerank의 전체 작업 처리 시간 비율을 제시한다. 데이터는 두 워크로드 모두 5GB를 사용했다. 스테이지간 처리 단계 최적화를 적용한 경우 입력 데이터를 RDD로 만들 때 각 파티션마다 같은 키 값을 갖는 엘리먼트들이 집중돼서 분포하기 때문에 스테이지간 처리 단계에서 소팅 및 부가적인 지연시간이 감소한다. 이에 대한 결과로 WordCount는 기존대비 77%로 처리 시간이 감소하고, Pagerank는 기존대비 86%로 처리 시간이 감소했다.

5.4 파티셔닝 최적화 성능

Fig. 7에서는 기존의 3가지 워크로드들에 대한 성능과 파티셔닝 최적화가 적용됐을 때의 성능을 비교한다. 파티셔닝 최적화를 적용했을 때 엑세큐터의 작업 처리시간이 평준화 됐는지를 확인하기 위해 모든 엑세큐터들의 작업처리 시간에 대한 표준편차를 측정했다. 그리고 모든 엑세큐터들의 작업처리량이 평준화됨에 따라 실제 작업 완료 시간이 감소했는지를 확인하기 위해 엑세큐터들의 평균 작업 처리 시간을 측정했다. 엑세큐터들의 평균 작업 처리 시간은 기존대비 각각 85%, 87%, 77%로 감소했고, 엑세큐터들의 작업 완료 시간 표준편차는 최소 20.799(Pagerank)에서 최대 303.83 (Kmeans)만큼 감소했다. 결과적으로 WordCount와 Pagerank는 스테이지 내부 처리 최적화에 의한 처리시간 감소가 가장 크고, Kmeans는 파티셔닝 최적화에 의한 처리시간 감소가 가장 크다.

5.5 최적화 기법 조합들의 성능

Fig. 8에서는 기존 WordCount, Pagerank의 성능과 스테이지 내부 및 스테이지간 처리성능 최적화가 모두 적용됐을 때의 성능을 비교한다.

두 기법을 모두 적용할 경우 WordCount는 처리시간이 기존 대비 47%로 감소했고, Pagerank는 68%로 감소했다. 이는 각각의 기법을 적용했을 때보다 더 적은 처리시간으로 두 기법이 상호간에 부정적인 간섭없이 각각의 처리 단계에 독립적으로 적용되는 것을 입증한다.

Fig. 9는 기존 WordCount, Pagerank, Kmeans의 성능과 스테이지 내부 처리 최적화와 파티셔닝 최적화가 모두 적용된 WordCount, Pagerank의 성능을 비교한다. 두 가지 기법을 모두 적용할 경우 WordCount는 처리 시간이 기존대비 39%로, Pagerank는 56%로, Kmeans는 56%로 감소했다. 이는 스테이지

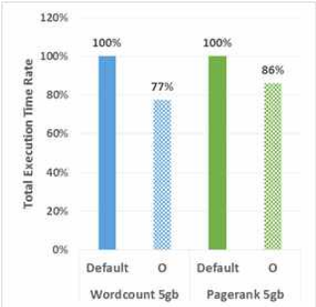

Fig. 6. The Execution Performance of "O" optimization

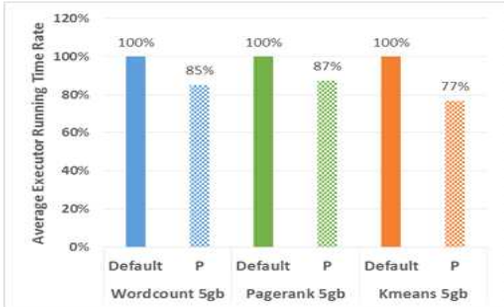

Fig. 7. The Execution Performance of "P" Optimization. The Right Graph Compares Standard Deviation of the Executor's Processing Time between Default and "P" Optimization

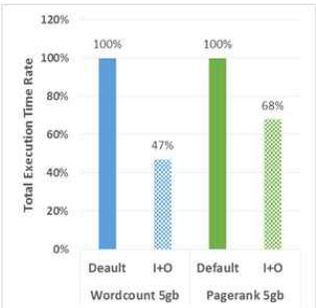

Fig. 8. The Execution Performance of "I+O" Optimization

Fig. 9. The Execution Performance of "I+P" Optimization. The Right Graph Compares Standard Deviation of the Executor's Processing Time between Default and "I+P" Optimization

내부 처리 단계와 시스템 설정 단계에서의 최적화가 간섭하더라도 더 좋은 시너지를 낸다는 것을 의미한다. 특히 Kmeans의 경우 스테이지 내부 처리 단계만 적용했을 때보다 월등히 처리시간이 감소하는 사실을 확인할 수 있다. 또한 엑세큐터들의 평균 처리시간과 이에 대한 표준편차도 Pagerank를 제외하면 파티셔닝 최적화만 적용했을 때보다 두 기법을 모두 다 적용했을 때 더 감소했다.

Fig. 10은 기존 WordCount와 Pagerank에 스테이지간 처리 최적화와 파티셔닝 최적화를 모두 적용했을 때 성능을 나타낸다. 결과적으로 각각의 기법을 적용했을 때보다 작업 처리시간, 평균 엑세큐터 작업처리시간, 엑세큐터 작업처리시간 표준편차 모두가 감소하는 사실을 확인할 수 있다. 3가지 기법들 중 2가지 기법들의 조합에 대한 성능은 스테이지 내부 처리 최적화와 파티셔닝 최적화의 조합, 스테이지 내부 처리 최적화와 스테이지간 처리 최적화의 조합, 스테이지간 최적화와 파티셔닝 최적화의 조합 순으로 성능 향상이 큰 것을 확인할 수 있다. 이런 결과는 성능 최적화 기법들이 조합이 되더라도 각각의 성능최적화 기법의 성능이 대체적으로 유지된다는 것을 의미한다. 결과적으로 이 실험들을 통해 각 최적화 기법들이 함께 적용될 경우 하나의 최적화 기법을 적용하는 것 보다 작업 처리 단계마다 처리 성능을 최적화 할 경우 최적화 기법간 간섭이 발생하더라도 더 큰 성능향상이 발생하는 것을 입증할 수 있었다.

Fig. 11은 기존 WordCount, Pagerank와 모든 최적화기법들을 적용했을 때의 성능을 나타낸다. 실험 결과를 보면 모든 기법들을 적용했을 때 기존보다 작업 처리 시간, 평균 엑세큐터 작업 처리 시간, 엑세큐터 작업 처리 시간 표준편차가 감소하지만 스테이지 내부 처리 최적화와 파티셔닝 최적화 조합의 성능과 유사한 것을 확인할 수 있다. 이는 3가지 기법을 모두 사용할 경우 WordCount, Pagerank 모두에서 최적화 기법들의 간섭현상이 있다는 것을 의미한다. 이에 대한 정확한 분석은 추후연구로 다룰 예정이다.

6. 관련 연구

아파치 스파크가 대중적인 분산 처리 프레임워크로 쓰이게 되면서 [11-15]와 같이 스파크의 성능을 향상시키기 위한 다양한 연구들이 진행됐다. 이들 중 대부분은 성능의 일부분을 개선하는데 그쳤다.

[16]은 작업의 시작부터 끝까지의 처리량을 개선하려고 노력했고, [17]은 네트워크 최적화를 통해 병렬 스트림 성능을 향상시키려고 노력했다. [10, 18]은 저장공간 최적화를 위해 인 메모리 캐시를 조율하는 연구를 수행했다. 이러한 연구들은 스파크의 특정 부분에서의 성능을 개선하는데 그쳤고 스파크의 성능에 영향을 주는 전반적인 요소들을 충분히 고려하지 못했다.

다른 선행연구들에서는 멀티 프로세서 시스템들[19], 클라우드 저장소 시스템들[20], 데이터베이스 시스템들[21, 22], 그래프 처리 시스템들[23, 24]의 저장소 입출력 성능과 처리 성능을 향상시키기 위해 데이터 파티셔닝 문제를 최적화하는 연구를 수행했다. 그러나 이러한 연구들은 직접적으로 스파크에 적용되기는 어렵다는 단점을 갖는다.

Fig. 10. The Execution Performance of "P+O" Optimization. The Right Graph Compares Standard Deviation of the Executor's Processing Time between Default and "P+O" Optimization

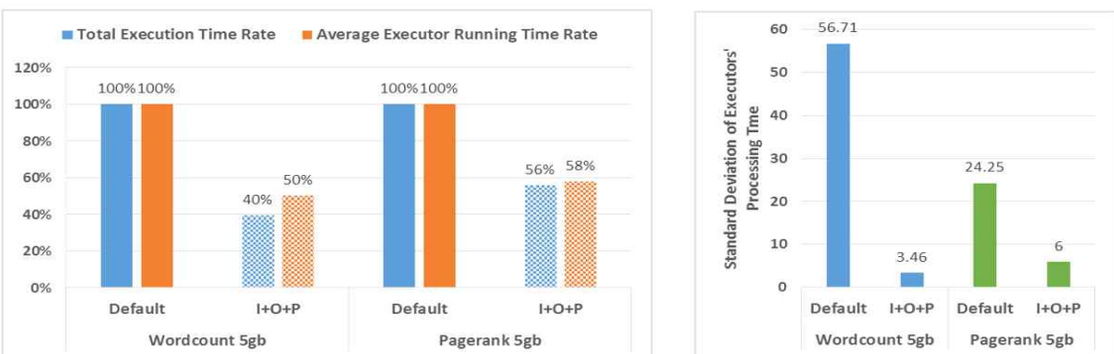

Fig. 11. The Execution Performance of "I+O+P" Optimization. The Right Graph Compares Standard Deviation of the Executor's Processing Time between Default and "I+O+P" Optimization

[25]는 스파크의 병렬화 수준을 최적화시킴으로써 처리 성능을 향상시키려고 했지만 워크로드와 저수준의 오퍼레이션들을 고려하지 못한 정적인 데이터 파티셔닝 기법을 개발했다. 또한 파라미터들의 튜닝이 어떤 관점에서 스파크의 성능에 영향을 주는지도 밝혀내지 못했다. [26]은 스파크의 처리과정 중 각각의 스테이지마다 다른 파티셔닝 수준을 적용하는 방법을 통해 병렬화 수준과 처리시간과의 상관관계를 연구했다. 그러나 일반적인 상황에서는 발생하지 않는 성능 결과와 본인들이 제안한 기법을 비교함으로써 연구 결과의 신뢰성을 떨어뜨렸다.

7. 결론 및 추후연구

빅 데이터에 대한 분석이 중요해지면서 분산 처리 시스템을 통한 데이터 분석이 매우 중요해졌다. 이러한 데이터 분석에서는 프로그래밍 난이도가 비교적 낮고 다양한 유형의 분석을 지원하는 분산처리 프레임워크를 사용한다. 그러나 분산처리 프레임워크에서 DAG 기반 분산 병렬처리 작업은 최적화 유무에 따라 성능차이가 매우 크고, 다양한 요소에 의해서 성능에 영향을 받기 때문에 시스템과 작업 모두를 고려해서 성능최적화를 하는 것은 매우 도전적인 일이다. 우리는 본 연구를 통해 아파치 스파크 클러스터의 구조와 DAG 기반 작업이 스파크에서 어떤 단계들을 거쳐 처리가 되는지에 대해 분석했다. 또한 서로 다른 작업 처리 단계들에 대해 3가지의 성능최적화 기법을 제시하고 다양한 관점에서 성능 향상치를 제시했다. 마지막으로 최적화 기법들을 조합해서 적용했을 때 다양한 관점에서 성능 향상치를 제시하고, 각각의 최적화 기법이 조합이 되더라도 성능 향상 효율이 유지되는 것을 입증했다. 추후에는 보편적인 데이터에 대해서도 제안한 최적화 기법들의 적용해서 기법들의 보편성을 입증할 것이다. 또한 데이터 분석 앱을 사용할 때마다 시스템설정 및 최적화 기법들과 관련된 파라미터들의 로그를 머신러닝 기법에 적용하여 자동적으로 시스템을 최적화하는 기법에 대해서 연구할 것이다.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," In *Proc. USENIX OSDI*, 2004.
[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," In *Proc. USENIX Hot- Cloud*, 2010.
[3] "Apache Flink" [Internet], http://flink.apache.org
[4] "Spark Streaming" [Internet], https://spark.apache.org/streaming/
[5] "Spark SQL" [Internet], https://spark.apache.org/sql/

[6] S. Venkataraman et al., "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics," in *Proceedings of the 13th USENIX conference on Networked Systems Design and Implementation*, 2016.
[7] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, Vol.29, No.12, pp. 1170-1183, 1986.
[8] J. Dean et al., "Large scale distributed deep networks," Advances in *Neural Information Processing Systems*, pp. 1223-1231, 2012.
[9] M. Isard et al., "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, pp.59-72, 2007.
[10] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pp. 2-2, 2012.
[11] A. Alexandrov et al., "MapReduce and PACT - comparing data parallel programming models," in Proceedings of the 14th Conference on Database Systems for BTW. Bonn, Germany: GI, pp. 25 - 44, 2011.
[12] J. Shi et al., "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proc. VLDB Endow.*, Vol.8, pp. 2110-2121, Sep., 2015.
[13] D. Warneke and O. Kao, "Exploiting dynamic resource allocation for efficient parallel data processing in the cloud," *IEEE Trans. Parallel Distrib. Syst.*, Vol.22, No.6, pp.985-997, Jun., 2011.
[14] M. Armbrust et al., "Scaling spark in the real world: The performance and usability," *Proc. VLDB Endow.*, Vol.8, pp. 1840-1843, Aug., 2015.
[15] A. Alexandrov et al., "The stratosphere platform for big data analytics," *The VLDB Journal*, Vol.23, No.6, pp.939-964, Dec., 2014.
[16] E. Yildirim and T. Kosar, "Network-aware end-to-end data throughput optimization," in *Proceedings of the First International Workshop on Network-aware Data Management*, NY, USA, pp.21-30, 2011.
[17] T. J. Hacker et al., "Adaptive data block scheduling for parallel tcp streams," in *HPDC-14. Proceedings. 14th IEEE International Symposiu on HPDC.*, pp.265-275, 2005.
[18] H. Li et al., "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC. New York: ACM, pp.6:1-6:15, 2014.
[19] J. Cieslewicz and K. A. Ross. "Data partitioning on chip multiprocessors," In *Proc. ACM Data Management on New Hardware*, 2008.
[20] C. Selvakumar, G. J. Rathanam, and M. R. Sumalatha. "Pdds - improving cloud data storage security using data partitioning technique," In *Proc. IEEE International Advance Computing Conference*, 2013.
[21] R. Nehme and N. Bruno, "Automated partitioning design in parallel database systems," In *Proc. ACM SIGMOD*, 2011.
[22] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems," In *Proc. ACM SIGMOD*, 2012.
[23] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," In *Proc. ACM EuroSys.*, 2015.
[24] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. "Graphx: Graph processing in a distributed dataflow framework," In *Proc. USENIX OSDI*, 2014.
[25] O. Marcu, A. Costan, G. Antoniu, and M. Pérez-Hernández, "Spark versus flink: Understanding the performance in big data analytics frameworks," In *Proc. IEEE Custer*, 2016.
[26] A. Paul, W. Zhuang, L. Xu, M. Li, M. Rafique, and A. Butt, "CHOPPER: Optimizing Data Partitioning for In-memory Data Analytics Frameworks," In *Proc. IEEE Cluster*, 2016.

명 노 영
http://orcid.org/0000-0002-5858-1667
e-mail : mry1811@korea.ac.kr
2014년 고려대학교 컴퓨터교육과(이학사)
2014년~현 재 고려대학교 컴퓨터학과 박사과정
관심분야 : 분산 컴퓨팅, 빅 데이터 분석, 분산 학습 알고리즘

유 헌 창
http://orcid.org/0000-0003-2216-595X
e-mail : yuhc@korea.ac.kr
1989년 고려대학교 전산과학과(이학사)
1991년 고려대학교 컴퓨터학과(이학석사)
1994년 고려대학교 컴퓨터학과(이학박사)
1995년~1998년 서경대학교 컴퓨터공학과 조교수
1998년~현 재 고려대학교 컴퓨터학과 교수
관심분야 : 분산 컴퓨팅, 클라우드 컴퓨팅

최 수 경
http://orcid.org/0000-0001-8667-8178
e-mail : csukyong@korea.ac.kr
1999년 부산여자대학교 컴퓨터교육과 (이학사)
2006년 고려대학교 컴퓨터교육과 (교육학석사)
2013년 고려대학교 컴퓨터교육과(이학박사)
2013년~현 재 고려대학교 정보창의교육연구소 연구교수
관심분야 : 그리드 컴퓨팅, 클라우드 컴퓨팅, 소셜 클라우드 컴퓨팅