2020년도
학사학위논문

아파치 스파크 클러스터를 사용한 지리정보 빅데이터 분석

: 행정구역별 대중교통 이용량 분석

Big Data Analysis for Geographic Information using Apache Spark Cluster

: Analysis of Public Transportation Utilization by Administrative Region

2020년 11월 28일

순천향대학교 공과대학
컴퓨터공학과

김혜진

2020년도
학사학위논문

아파치 스파크 클러스터를 사용한 지리정보 빅데이터 분석

: 행정구역별 대중교통 이용량 분석

Big Data Analysis for Geographic Information using Apache Spark Cluster

: Analysis of Public Transportation Utilization by Administrative Region

2020년 11월 28일

순천향대학교 공과대학
컴퓨터공학과

김혜진

아파치 스파크 클러스터를 사용한 지리정보 빅데이터 분석

: 행정구역별 대중교통 이용량 분석

Big Data Analysis for Geographic Information using Apache Spark Cluster

: Analysis of Public Transportation Utilization by Administrative Region

지도교수 이 상 정

이 논문을 공학사학위 논문으로 제출함

2020년 11월 28일

순천향대학교 공과대학
컴퓨터공학과

김혜진

김혜진의 공학사학위논문을 인준함

2020년 11월 28일

심 사 위 원 이 상 정 인

지 도 교 수 이 상 정 인

순천향대학교 공과대학

컴퓨터공학과

초 록

최근 몇 년간 화두가 되고 있는 4차 산업혁명은 우리 생활에 많은 변화를 가져올 것으로 예상된다. 4차 산업혁명은 사물 간의 초연결성과 연결된 정보를 활용하는 초지능 사회를 근간으로 한다. 이러한 초연결성과 초지능은 사회·경제적으로 빠른 속도로 넓은 범위에 영향을 끼치게 될 것이다. 미래의 산업수요는 데이터 및 아이디어가 집적되는 도시공간에서 발생할 것이며, 도시는 하나의 플랫폼 역할을 수행 하게 될 것이다. 여기서 주목해야 할 것은 4차 산업혁명의 핵심 기술 들이 국토, 도시, 지역 등 공간에 집중된다는 것이다.

특히 이러한 정보를 분석하고 결합하면 의미 있는 결과를 얻을 수 있다는 사실이 알려지면서 다양한 요구들이 등장함에 따라 누적되는 정보는 기하급수적으로 증가하게 되었다. 최근 새로운 형태의 대규모 공간 데이터들의 저장과 분석을 위해 다양한 공간 빅 데이터 기술들이 제시되고 있다. 더 나아가서 GPS가 내장된 모바일 기기들의 사용이 증가함에 따라 위치정보가 태그된 소셜 데이터와 사진, 장소 로깅 정보 등 공간 정보를 포함하는 컨텐츠가 계속적으로 축적되고 있으며, 분석되어야할 중요한 대상으로 등장하고 있다. 이와 같은 데이터의 증가는 새로운 데이터 분석 시스템을 요구하게 되었고, 최근 위와 같은 대규모 데이터의 분석을 위한 분산처리 시스템이 대두되고 있다.

공간 정보의 생성량 증가는 공간 키워드 질의에서 사용해야 할 데이터의 증가로 이어졌고, 이는 기존의 공간 키워드 질의 기법들이 대용량의 공간 키워드 질의를 처리하는데 문제점으로 작용한다. 따라서 대용량의 데이터를 빠르게 검색 및 분석하기 위해서는 분산 처리 환경에서 공간 키워드 질의를 지원하는 방법이 요구된다.

이를 위해 분산 컴퓨팅 프레임워크인 아파치 스파크(Apache Spark)에서 공간 데이터 처리를 가능하게 하는 아파치 세도나(Apache Secona)를 사용한다. 세도나는 스파크에서 직접 사용이 불가능한 공간 쿼리의 단점을 보완 할 수 있다는 이점이 있다. 따라서, 본 논문은 지리정보 빅 데이터를 처리하기 위한 메모리 내 클러스터 컴퓨팅 시스템인 아파치 세도나 사용에 의의를 둔다.

주요어 : 지리정보, 공간정보, 공간데이터, 빅데이터, 대용량, 아파치 스파크(Apache Spark), 아파치 세도나(Apache Secona), 클러스터

차 례

제 1 장 서 론 ····· 1

제 2 장 이론적 배경 ····· 5

2.1 공간데이터(Spatial Data) ····· 5

2.1.1 공간데이터 정의 ····· 5

2.1.2 공간데이터 필요성 ····· 6

2.2 클러스터 컴퓨팅(Cluster Computing) ····· 7

2.3 아파치 하둡(Apache Hadoop) ····· 8

2.4 아파치 스파크(Apache Spark) ····· 10

2.5 아파치 제플린(Apache Zeppelin) ····· 13

제 3 장 지리정보 빅데이터 분석 시스템 ····· 14

3.1 지리정보 빅데이터 분석 시스템 설계 ····· 14

3.1.1 아파치 세도나(Apache Sedona) ····· 16

3.1.2 지리정보 빅데이터 분석 시스템 제안 ····· 23

3.2 지리정보 빅데이터 분석 시스템 환경 구축 ····· 25

3.2.1 지리정보 빅데이터 분석 시스템 클러스터 환경구축 ····· 27

3.2.2 아파치 세도나 환경 구축 ····· 35

제 4 장 공간 연산 성능 평가 ····· 36

4.1 공간 연산 환경 구성 ····· 36

4.2 공간 연산 진행 방법 ····· 37

4.2.1 전처리 ····· 37

4.2.2 공간 연산 적용 ····· 42
4.3 공간 연산 성능 평가 ····· 48
4.3.1 단일머신 ····· 48
4.3.2 클러스터 ····· 50

제 5 장 결론 및 향후 계획 ····· 54

참고문헌 ····· 56

감사의 글 ····· 57

표 차 례

[표 1] 노드 구성 사양정보 및 툴 ····· 26

그 림 차 례

[그림 1] 디스크처리와 램 처리 ····· 11
[그림 2] 스파크 라이브러리 구성 ····· 12
[그림 3] 제플린 시각화 ····· 13
[그림 4] 지리정보 빅데이터 분석 시스템 구성도 ····· 14
[그림 5] 아파치 세도나의 구조 ····· 16
[그림 6] 지리정보 빅데이터 분석 시스템 처리 과정 ····· 24
[그림 7] 노드 구성 ····· 25
[그림 8] 호스트 파일 설정 ····· 27
[그림 9] 하둡 파일 시스템으로 옮겨준 세도나 jar 파일 목록 ····· 35
[그림 10] 스키마 생성, 데이터 프레임, SRDD 및 뷰 등록 ····· 38
[그림 11] 시간대별 버스 승하차 데이터 캐시 등록 ····· 38
[그림 12] 시간대별 버스 승객 승하차 전처리 함수 ····· 39
[그림 13] 승하차 전처리 함수 출력 화면 ····· 40
[그림 14] 버스 정류장 좌표 데이터 ····· 40
[그림 15] 경계 좌표 데이터 ····· 41
[그림 16] 공간 범위 질의를 사용한 SQL ····· 42
[그림 17] 공간 KNN 질의를 사용한 SQL ····· 44
[그림 18] 공간 조인 질의를 사용한 SQL ····· 46
[그림 19] 단일머신에서의 공간 질의 연산 처리 ····· 49
[그림 20] 클러스터 4개를 사용한 공간 질의 연산 처리 ····· 50
[그림 21] 클러스터 8개를 사용한 공간 질의 연산 처리 ····· 51
[그림 22] 클러스터 12개를 사용한 공간 질의 연산 처리 ····· 51
[그림 23] 클러스터 실험 종합 ····· 52

제 1 장 서 론

최근 몇 년간 화두가 되고 있는 4차 산업혁명은 우리 생활에 많은 변화를 가져올 것으로 예상된다. 4차 산업혁명은 사물 간의 초연결성과 연결된 정보를 활용하는 초지능 사회를 근간으로 한다. 이러한 초연결성과 초지능은 사회·경제적으로 빠른 속도로 넓은 범위에 영향을 끼치게 될 것이다. 미래의 산업수요는 데이터 및 아이디어가 집적되는 도시공간에서 발생할 것이며, 도시는 하나의 플랫폼 역할을 수행 하게 될 것이다. 여기서 주목해야 할 것은 4차 산업혁명의 핵심 기술 들이 국토, 도시, 지역 등 공간에 집중된다는 것이다.

이에 따라 우리 정부에서도 드론, 자율주행차, 공간정보, 스마트시티, 제로에너지빌딩, 리츠 등이 포함된 7대 신산업을 육성하겠는 계획을 수립한 바 있다. 특히 공간정보는 다른 분야의 인프라로서의 역할과 자체적인 산업 육성을 통해 경제 활성화에 기여할 수 있다는 점 때문에 더욱 중요하다고 볼 수 있다.

일상생활에서 쉽게 활용되는 공간정보는 다양하게 분포한다. 일례로 스마트폰 애플리케이션 중에서 약 80% 정도가 공간정보를 포함하고 있다. 공간정보 활용의 중요성은 정부 정책에서도 찾아볼 수 있다. 정부는 공공정보 개방 정책을 추진하면서 국가 공간정보포털, 국가공간정보오픈플랫폼 등을 다양한 서비스를 통해 생산한 공간정보를 민간에 제공하고 있다.

그렇지만 이러한 공공정보만으로는 충분하지 않아 보인다. 새로운 공간정보에 대한 욕구가 점점 커지고 있기 때문이다. 현재 다양한 센서(사람 혹은 차량의 이

동정보)정보, 통계정보, 신용카드 매출 정보, CCTV 정보 등 여러 가지 형태의 정보들이 디지털의 형태로 누적·저장되고 있다. 이러한 정보는 공간정보와 결합하면서 다양하게 활용될 수 있다. 택시나 버스 또는 지하철 승하차 정보를 1년간 모아놓은 정보나 부동산실거래가 정보 등 거래정보를 공간정보와 융합하여 표현하는 방식이나 드론과 자율주행차, 로봇 등이 운행하기 위해 공간정보를 기존에 축적한 데이터와 결합하여 활용하는 방식이 주목을 받고 있다.

특히 이러한 정보를 분석하고 결합하면 의미 있는 결과를 얻을 수 있다는 사실이 알려지면서 다양한 요구들이 등장함에 따라 누적되는 정보는 기하급수적으로 증가하게 되었다. 최근 새로운 형태의 대규모 공간 데이터들의 저장과 분석을 위해 다양한 공간 빅 데이터 기술들이 제시되고 있다. 더 나아가서 GPS가 내장된 모바일 기기들의 사용이 증가함에 따라 위치정보가 태그된 소셜 데이터와 사진, 장소 로깅 정보 등 공간 정보를 포함하는 컨텐츠가 계속적으로 축적되고 있으며, 분석되어야할 중요한 대상으로 등장하고 있다. 이와 같은 데이터의 증가는 새로운 데이터 분석 시스템을 요구하게 되었고, 최근 위와 같은 대규모 데이터의 분석을 위한 분산처리 시스템이 대두되고 있다. 초기의 공간 빅 데이터 시스템들은 대부분 하둡(Hadoop) 프레임워크(Framework) 기반의 맵리듀스(MapReduce)를 이용하여 구현되었다.

실제로 기존의 관계형 DBMS를 확장한 공간 데이터베이스 시스템은 공간 데이터의 규모가 커짐에 따라 확장성(scalability) 이슈의 문제가 대두되었으며, 최근에는 하둡 에코 시스템(Hadoop ecosystem)을 확장하여 대규모로 공간 분석을 수행 바 있다. 이들 연구는 확장성에 대한 문제는 해결하였으나 여전히 수행 시간이 길어 성능이 좋지 않았다.

위와 같은 시스템에서 사용하는 맵리듀스는 빅 데이터내의 각 데이터를 필터링 하거나 변환하는 맵 과정과 그 대규모 결과들을 통합하여 결과를 만들어내는 리듀스 과정으로 구성되어 있다. 복잡한 빅 데이터 분석을 위해서는 다수의 맵 과정 리듀스 과정이 복합적으로 사용되는데, 이때, 한 번의 과정이 종료될 때마다 생성된 결과 데이터는 디스크에 저장을 한다. 이러한 과정이 디스크 I/O 시간을 늘어나는 단점으로 작용한다. 이는 하둡 맵리듀스기반 시스템들의 공간 빅 데이터 시스템들에도 그대로 존재하며, 이러한 단점으로 인해 실시간 서비스와 같은 빠른 데이터 처리를 요구하는 서비스에서는 사용하기 매우 어렵다. 따라서 빠른 데이터 처리를 위하여 인-메모리 데이터 처리 시스템이 필요하다.

이를 위해 하둡의 문제점을 극복하기 위하여 인-메모리 기반의 스파크 시스템을 활용하는 공간 빅 데이터 시스템들이 개발되었다. 관련 시스템에는 Sedona, SpatialSpark, LocationSpark 등이 있다. 스파크를 활용하는 시스템들은 메모리에서 데이터를 저장 및 처리를 하고, 중간 과정의 데이터를 저장할 필요가 없기 때문에 하둡의 단점을 극복하였다. 하지만 하둡을 활용하는 시스템이나 스파크를 활용하는 시스템 모두 공간 키워드 질의를 지원하지 않고 있다.

공간 정보의 생성량 증가는 공간 키워드 질의에서 사용해야 할 데이터의 증가로 이어졌고, 이는 기존의 공간 키워드 질의 기법들이 대용량의 공간 키워드 질의를 처리하는데 문제점으로 작용한다. 따라서 대용량의 데이터를 빠르게 검색 및 분석하기 위해서는 분산 처리 환경에서 공간 키워드 질의를 지원하는 방법이 요구된다.

본 논문에서는 위와 같은 문제점을 해결하기 위하여 스파크 클러스터 상에서 대용량의 공간 키워드 질의를 지원하는 공간 분산 프레임워크 아파치 세도나(Apache Secona)를 제안하고, 이를 활용한 시스템을 설계 및 구현한다. 제안하는 프레임워크는 다음의 특징을 갖는다.

- 메모리 기반의 분산처리 환경에서 공간 키워드 질의를 지원하는 프레임워크인 세도나를 제안한다. 제안하는 프레임워크는 인-메모리 환경에서 분산 처리를 지원하는 스파크를 기반으로 하였다. 세도나에는 공간 인덱스와 공간 키워드 인덱스 추가가 가능하며, 여러 가지의 분할 전략을 추가할 수 있게 함으로써, 여러 응용프로그램으로 확장할 수 있다.

본 논문은 다음과 같이 구성 한다. 2장에서는 본 논문에 사용되어지는 공간 데이터의 개념과 오픈 소스 소프트웨어에 대하여 설명한다. 3장에서는 지리정보 빅데이터 분석 시스템에 대하여 설명한다. 4장에서는 분할 분산 처리 환경에서 사용하는 공간 분할 알고리즘에 대해서 알아보며, 시스템에 대한 공간 연산 성능 평가를 한다. 마지막으로 5장에서는 논문의 끝을 맺는다.

제 2 장 이론적 배경

본 장에서는 본 연구를 위해 조사하였던 공간데이터에 관한 개념과 클러스터 컴퓨팅 및 아파치 하둡, 아파치 스파크, 아파치 세도나, 아파치 제플린에 관한 연구들에 대해 정리하고자 한다.

이 연구들을 통해 공간데이터에 대한 개념과 더불어 클러스터 컴퓨팅 및 하둡, 스파크, 세도나, 제플린에 대한 기본 동작과 전체 시스템 구조를 이해할 수 있었다.

2.1 공간데이터(Spatial Data)

2.1.1 공간 데이터 정의

공간정보는 데이터의 형태에 따라 도형 데이터와 속성 데이터로 구성된다. 또한 정보의 단위를 기준으로는 공간정보를 국토공간정보와 도시공간정보로도 구분할 수 있다. 공간정보를 국가단위로 볼 때에는 국토공간정보라 하며, 지형, 지질, 토지이용, 자연환경, 통계 데이터 등이 이에 해당된다. 도시규모에서는 도시공간정보라고 하며 도로, 토지, 가옥, 상 · 하수도, 가스, 전기공급시설 등이 이에 포함된다. 이러한 공간정보를 생산, 관리, 가공, 유통, 활용하거나 다른 정보기술과 융합해 시스템을 구축하고 관련 서비스를 제공하는 일련의 산업을 공간정보산업이라고 한다.

2.1.2 공간 데이터 필요성

인류가 출현하여 자급자족으로 생활을 유지해 오던 농업사회의 핵심 기술 및 요소는 일할 수 있는 체력 및 토지, 근면한 정신이었으며, 자신이 소유한 토지 경계를 확정하는 것이 무엇보다 중요하였다. 이러한 농업사회의 특성상 자연재해에 따른 재산적 손실을 막는 것이 가장 큰 경영전략이었다. 따라서 이러한 정보(토지 경계, 자연재해 피해 지역 등)를 제공할 수 있는 지도의 확보가 무엇보다 중요하였다.

산업사회를 통해 이룩된 컴퓨터, 정보통신 기술 등의 발전으로 인해 사회는 지식과 정보가 핵심 기술이 되는 정보사회로 변화하게 되었다. 즉, 개인이나 국가가 가지고 있는 지식과 정보가 사회를 움직이게 되었으며 정보통신기술을 이용하여 개인이 가지고 있는 지식을 공유하고 개방하는 것이 수월하게 되었다. 따라서 다양한 정보를 이용하여 가장 최적의 계획을 수립하고 이를 수행하는 것이 핵심가치가 되었다. 따라서 정보사회에서는 인간의 행동양식을 결정하는데 있어서 공간정보가 가장 핵심이 되는 정보로 부각되었다.

이러한 정보사회를 토대로 다가올 미래사회는 지식정보사회에서 스마트 사회로 변하게 될 것으로 예상된다. 스마트 공간정보사회는 모든 영역에 걸쳐 국가 전반에 구축된 공간정보 및 IT 인프라와 다양한 모바일 기기를 통해 수집된 모든 정보를 수집하고 가공, 활용함으로써 현실을 거의 있는 그대로 재창조 할 수 있을 것이다.

이상의 사회 패러다임 변화에서 살펴본 바와 같이 공간정보는 사회를 이끌어

가기 위해 없어서는 안 될 가장 중요한 정보이다. 따라서 국가 전반에 걸쳐 구축된 공간정보 인프라는 국가 경쟁력을 좌우 할 수 있으며, 국민의 삶의 질을 나타내는 척도로 사용될 수 있다.

2.2 클러스터 컴퓨팅(Cluster Computing)

컴퓨터 클러스터는 연결된 컴퓨터 그룹이며, 그룹 내 컴퓨터 서버 간에 서로 긴밀하게 협력 작동하는 병렬 또는 분산 처리 시스템이다. 컴퓨터 서버는 메모리 I/O 기능 및 운영 체제가 단일 또는 다중 프로세서 시스템 (PC 또는 워크 스테이션) 이다. 클러스터는 일반적으로 서로 연결된 두 개 이상의 컴퓨터 (서버)를 나타낸다. 서버는 단일 캐비닛에 존재하거나 LAN을 통해 물리적으로 분리되어 연결될 수 있다.

일반적으로 클러스터는 단일 컴퓨터에서 제공되는 성능 및/또는 가용성을 향상 시키기 위해 배포되며 단일 컴퓨터에 비해 속도와 가용성 면에서 매우 경제적이다. 지난 10여년 동안 저비용 컴퓨터를 연결하여 응용 프로그램을 처리하는 클러스터 기술이 개발되어져 왔다. 클러스터 컴퓨팅은 경제성, 성능 및 유연성으로 인해 중앙 집중식 컴퓨팅 모델에 대한 대안으로 개발되었다. 많은 기업들이 데이터 센터 내에 서 애플리케이션 성능, 가용성 및 확장성을 향상시키기 위해 고성능 저비용 컴퓨터 클러스터를 모색하고 있다. 컴퓨터 클러스터에 대한 관심과 사용은 주로 컴퓨터와 네트워크 스위치의 성능 향상, 그리고 응용프로그램의 발달로 이루어졌다. 클러스터 컴퓨팅은 하드웨어, 네트워크 및 소프트웨어를 통해 통합된 상용 컴퓨터와 자원을 하나의 컴퓨터로 통합한다.

클러스터의 주요 이점은 확장성, 가용성 및 성능이다. 확장성을 위해 클러스터는 컴퓨팅 서버의 결합된 처리 성능을 사용하여 단일 시스템이 제공할 수 있는 것보다 높은 성능으로 병렬 데이터베이스 서버와 같은 클러스터 지원 응용 프로그램을 실행한다. 클러스터의 처리 능력을 확장하는 것은 클러스터에 서버를 추가만 하면 된다. 장애가 발생하면 클러스터 내의 서버가 서로 백업을 제공하므로 클러스터 내의 가용성이 보장된다. 고가용성 클러스터에서 서버가 서비스를 중단하거나 실패하면 서버가 클러스터 내의 다른 서버로 전송된다. 실행중인 응용 프로그램 및 데이터는 실행 중인 서버에 장애가 발생하여도 다른 서버로 옮겨서 실행된다.

클러스터 컴퓨팅은 클러스터의 관리 편의성이 향상되어 사용자 관점에서는 응용 프로그램 자원을 서비스 및 응용 프로그램 공급자로 간주된다. 사용자는 자원이 단일 서버, 클러스터 또는 클러스터 내의 어떤 서버가 서비스를 제공하는지 알 수 없고 알 필요도 없다. 클러스터는 확장 가능한 용량, 다운 타임 없는 확장성, 작업 부하에서 예기치 않은 피크 처리 기능, 단일 시스템과 같은 중앙 관리, 중단 없는 가용성을 제공한다. 이러한 클러스터의 종류는 계산 집약 또는 데이터 집약 어플리케이션을 위한 분산/병렬 클러스터와 로드 균형조정 클러스터 등이 있다.

2.3 아파치 하둡(Apache Hadoop)

하둡은 구글이 발표한 GFS(Google File System) 와 맵리듀스(MapReduce)를 2005년에 더그 커팅(Doug Cutting) 이 구현한 결과물이다. 처음에는 루씬을 기반으로한 오픈소스 검색 엔진 너치(Nutch) 에 적용시키기 위해 개발 되었지만 이

후 독립적인 프로젝트로 만들어졌다. 하둡은 분산 파일 시스템 HDFS(Hadoop distributed file system)과 분산처리 프로그래밍 모델인 맵리듀스로 구성된다. 데이터를 분산 파일 시스템인 HDFS에 저장하고, 맵리듀스 라는 분산 처리 시스템을 사용하여 데이터를 처리한다. 하둡은 빅데이터를 처리하는 프로그램으로써 상호보완적인 프로그램들과 함께 하둡 생태계를 이루고 있다. 하둡은 하나의 마스터 서버와 여러 대의 슬레이브 서버를 지정하여, 각 서버에 데이터를 저장한다. 그리고 데이터가 저장된 각 서버에서 동시적으로 데이터를 처리한다. 하둡은 이러한 분산 컴퓨팅을 통해 기존의 데이터 분석 방법으로 뛰어난 성과를 보여준다.

하둡 분산 파일 시스템은 대용량 파일을 분산된 클러스터에 저장하고, 이 저장된 대용량 파일을 많은 사람들이 사용 가능하며, 범용 하드웨어에서 실행되도록 설계되었다. 내결함성이 높으며 저가의 하드웨어에 배치되도록 설계되었다. 하둡 파일 시스템은 문제의 발생을 빠른 시간 내에 감지하여 상황에 따라 재빠르게 처리 할 수 있다. 또한 스트리밍 방식의 데이터 접근으로 빠른 시간 내에 처리하는 것보다 같은 시간 내에 많은 데이터를 처리한다. 응용프로그램 데이터에 대한 높은 처리량의 액세스를 제공하며 대용량 데이터 세트가 있는 응용 프로그램에 적합하다.

맵리듀스는 구글에서 빅데이터를 분산 병렬 컴퓨팅 환경에서 처리하기 위해 만들어진 소프트웨어 프레임워크이다. 빅데이터를 병렬 처리 하기 위해 먼저 하나의 잡(Job)을 여러 개의 태스크(task)로 나누어 실행한다. 그 후 각각의 태스크는 서로 연관되어 있는 데이터를 수집하는 맵 과정과 필터링 과정에서 중복된 데이터를 없애고, 정리된 데이터들을 정렬 과정에서 정렬 시킨 후 필요한 데이터만을 추출하는 리듀스과정을 거친다. 맵리듀스는 여러 개의 작업을 한꺼번에 묶

고, 간단한 입출력을 갖는 알고리즘에 가장 적합화 되어있다. 따라서 복잡한 알고리즘을 구성하기에 부족하고, 반복적인 데이터를 처리함에 있어 성능의 제약을 받는다.

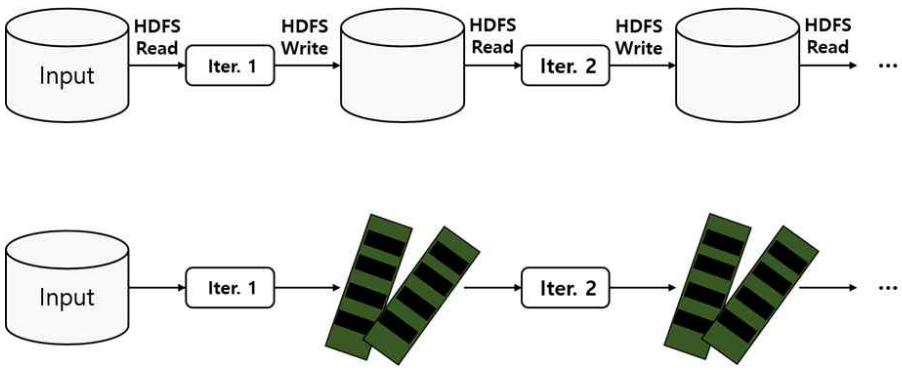
하둡은 2.0이 나옴으로써 기존의 하둡1.0에 비해 향상된 성능으로 보다 많은 클러스터와 HDFS의 고가용성 기능이 구현됐고, 리눅스뿐만 아니라 마이크로소프트의 윈도우도 지원한다. 더불어 맵리듀스 2.0으로도 불리는 얀(Yarn : yet another resource negotiator)이 포함됨으로써 맵리듀스 외에 여러 프로세싱 알고리즘을 쉽게 플러그인 시킬 수 있게 되어 맵리듀스에 의존하지 않는 다양한 데이터 프로세싱 엔진의 구동이 가능하게 되었다.

2.4 아파치 스파크(Apache Spark)

스파크는 2009년 버클리 캘리포니아 대학(University of California, Berkeley)의 AMPLab에서 자신들이 만든 Mesos의 개념을 증명하기 위한 프로젝트로 시작되었으며, 2013년에 아파치 소프트웨어 재단 부화형 프로젝트로, 2014년에는 재단 최상위 프로젝트 중 하나가 되었다.

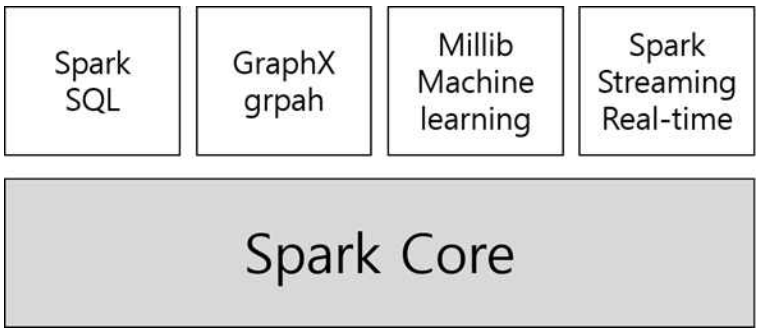
스파크는 범용의 고성능 클러스터 컴퓨팅 시스템이다. 스파크는 자바, 스칼라, C언어 그리고 Python 으로 작성된 상위 수준의 API를 제공하고, 일반적인 실행 그래프를 지원하는 최적화된 엔진이다. 또한, SQL과 구조화된 데이터 처리를 위한 Spark SQL, 기계 학습을 위한 MLlib, 그래프 처리를 위한 GraphX, 데이터 마이닝, 분석을 위한 SparkR, 그리고 Spark Streaming 과 같은 풍부한 고급 도구들을 지원한다.

[그림 1]과 같이 Spark는 Hadoop의 맵리듀스와 달리 디스크기반이 아닌 인메모리 기반의 소프트웨어로서 RDD(Resilient Distributed Datasets)라는 RAM상에서 읽기전용 데이터 구조를 기반으로 하고 있으며, 반복적인 RDD 연산을 통해 이루어진다. 이런 반복적인 RDD연산을 통해서 분산되어 있는 노드의 오류가 발생하여 데이터의 문제가 생겼을 경우, 앞서 반복되던 RDD반복에 대한 정보를 바탕으로 남아있는 연산을 다시 하여 문제를 해결해 간다.



[그림 1] 디스크처리와 램 처리

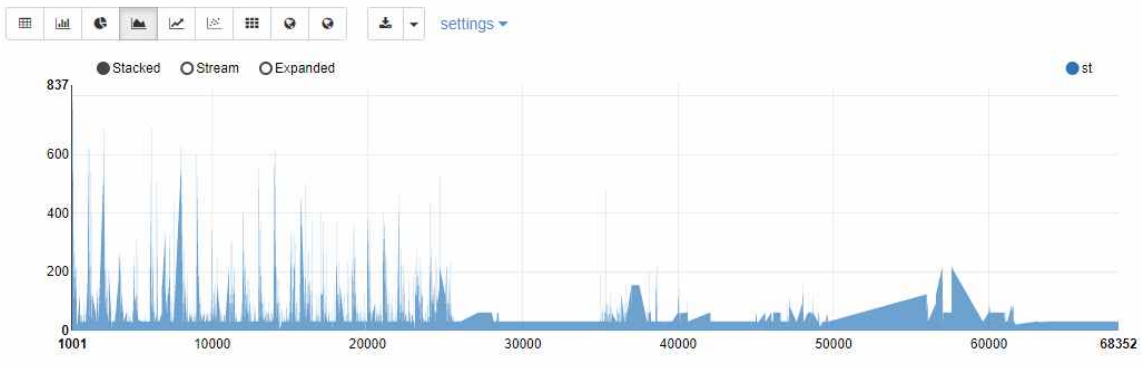
Spark는 [그림 2]과 같이 4가지의 대표적인 라이브러리로 구성되어 있다. 우선 첫 번째로 인메모리 기반의 RDD데이터셋에 SQL 쿼리를 가능하게 해주는 SpaarkSQL, 그래프이론·알고리즘을 수행할 수 있는 전문 라이브러리를 갖고 있는 GraphX엔진, 머신러닝 전문 라이브러리를 갖고 있는 MLlib 그리고 실시간 Real-time 데이터를 받고 처리할 수 있는 Spark Stream이 있다. 추가적으로 Spark 1.4.0버전이 나오면서 기존에 베타였던 R on Spark가 정식으로 라이브러리가 등록되면서 SparkR까지 총 5가지의 정식 하위 라이브러리가 존재한다.



[그림 2] 스파크 라이브러리 구성

2.5 아파치 제플린(Apache Zeppelin)

제플린은 스파크와 연동하기 쉬운 GUI(Graphic User Interface)를 제공하는 오픈소스 도구로서 인터프리터(Interpreter)방식의 명령과 수행을 제공한다. iPython과 같이 사용자가 수행하려는 코드나 결과 값 들을 저장하고 보여주어 노트북이라고도 불리는 포지션에 위치한다. 제플린은 html, sql, shell script를 비롯하여 scala/python/java 등을 지원하고 있으며 [그림 3]와 같은 화면으로 구성된다.



[그림 3] 제플린 시각화

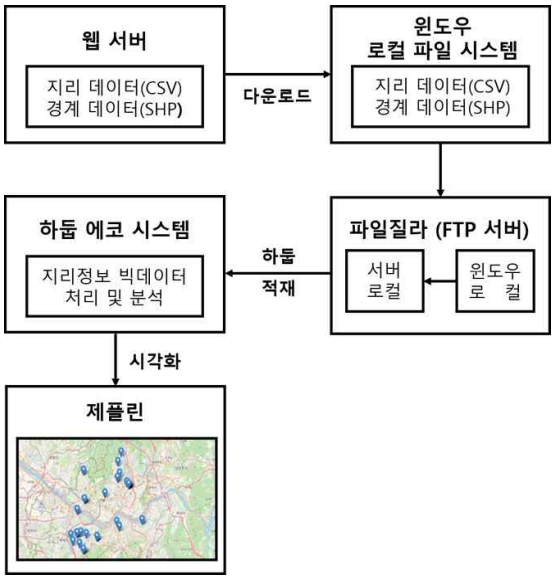
제플린은 사용자가 원하는 데이터를 추출, 처리, 분석을 손쉽게 처리 할 수 있도록 한다. 현재 아파치 인큐베이터에 등록되어 있으며 하둡의 대표적인 벤더 중 하나인 호튼웍스의 하둡 에코시스템에도 큰 관심을 보이고 있어, 앞으로 지속적인 개발과 성장이 기대되는 오픈소스 소프트웨어 중 하나이다.

제 3 장 지리정보 빅데이터 분석 시스템

본 논문에서 제안하는 아파치 스파크 클러스터를 이용한 지리정보 빅데이터 분석 시스템을 구축하기 위한 사항들을 설명하였다. 이 시스템을 기반으로 지리정보 빅데이터 수집, 분석, 처리 및 시각화 등 전체적인 구성 요소들을 설계 및 구현하였다.

3-1. 지리정보 빅데이터 분석 시스템 설계

본 논문에서 구현한 지리정보 빅데이터 분석 시스템은 데이터 다운로드, 적재, 처리, 분석 과정을 통해 시각화 되어 진다. 지리정보 빅데이터를 분석하기 위한 시스템 구성 개요는 아래 [그림 4]와 같다.

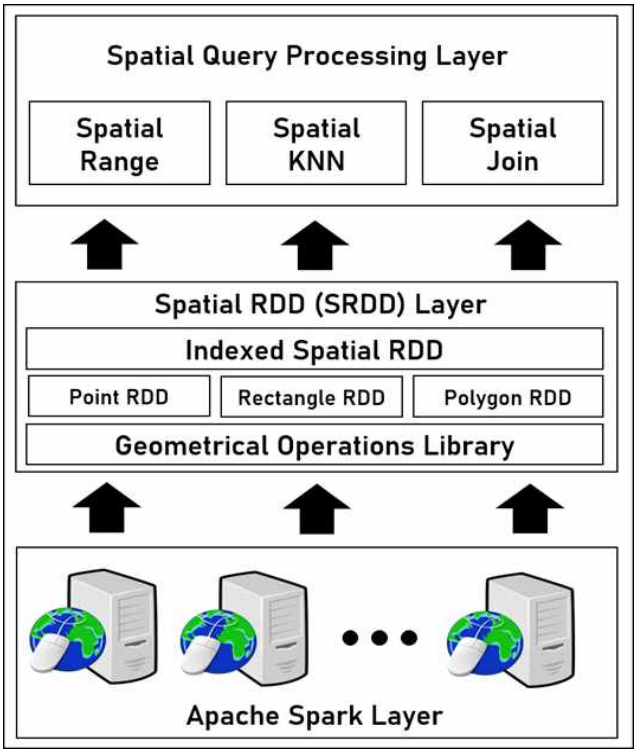


[그림 4] 지리정보 빅데이터 분석 시스템 구성도

분석에 필요한 데이터는 웹 서버로부터 다운로드하여 로컬 파일 시스템에 저장한다. 저장된 데이터는 FTP(File Transfer Protocol) 서버인 파일질라를 거쳐 하둡에 적재 된다. 적재된 데이터는 하둡 에코 시스템 내부에 존재 하는 프레임 워크들로 부터 처리 및 분석 과정을 거쳐 제플린을 통해 분석된 데이터를 시각화 할 수 있도록 하였다.

3.1.1 아파치 세도나(Apache Sedona)

RDD(Resilient Disributed Data)에 있는 원시 데이터를 공간정보를 포함하는 지오메트리(Geometry) 객체로 변환해주고, 지오메트리로 변형된 RDD에 공간 인덱스를 구축하여 공간 조인이나 KNN(K-Nearest Neighbor)같은 공간 객체 연산을 수행한다. 세도나는 [그림 5]와 같이 아파치 스파크 레이어(Apache Spark Layer), 공간 RDD 레이어(SRDD, Spatial RDD Layer), 공간 쿼리 처리 레이어(Spatial Query Processing Layer)로 구성되어 있으며, 이를 사용하기 위해 대표적으로 파티셔닝 및 인덱싱 작업이 이루어진다.



[그림 5] 아파치 세도나의 구조

아파치 스파크 레이어(Apache Spark Layer)

아파치 스파크 레이어는 기본 스파크를 이용하는 레이어 이다. 이는 세도나로부터 공간 데이터에 관한 모든 작업과 인덱스를 지원받아 공간 빅데이터를 처리할 수 있게 된다. 또한, 이 레이어는 스토리지(Storage) 에서 데이터를 영구적으로 읽고 저장하는 데에 있어 큰 역할을 한다.

공간 RDD 레이어(SRDD : Spatial RDD Layer)

공간 RDD 레이어는 스파크에서 읽어온 원시 데이터를 공간 데이터로 변경하여 PointRDD, PolygonRDD, RectangleRDD를 만들어준다. 즉, 각각의 RDD에 공간 인덱스를 생성하여 주는 것 이다. 또한, 세도나에서 전역 그리드를 사용하여 전체 영역을 나누고 각 그리드마다 지역 인덱스를 구축한다. 전역 그리드로 이진 분할 파티션을 사용하며 지역 인덱스로 R-tree나 Quad-tree를 사용한다.

공간 쿼리 처리 레이어(Spatial Query Processing Layer)

공간 쿼리 처리 레이어는 공간 데이터로 변경된 RDD에 위의 지역 인덱스를 구축하는 역할도 담당한다. 이때, 구축된 인덱스는 메모리에 상주시키고, 공간 쿼리 처리 레이어에서 사용된다. 공간 쿼리 처리 레이어는 사용자가 입력하는 공간 질의(범위질의, kNN 질의, 조인 질의등)는 공간 RDD 레이어에 있는 인덱싱된 공간 RDD(Indexed Spatial RDD)의 인덱스를 이용하여 수행된다.

SRDD 파티셔닝

기존의 스파크는 입력 데이터 파일을 메모리로 적재한 후에 같은 크기의 여러 파티션으로 분할하여, 각 파티션을 작업자 노드에 전달한다. 이러한 작업은 질의 속도 향상에 큰 영향을 미치지 못 하기 때문에 공간 인접성을 반영하지 못한다. 반면, 세도나는 적재된 SRDD를 내부 공간 데이터 분포에 따라 자동으로 다시 파티션한다. 즉, 공간 근접성에 기반하여 공간 데이터를 같은 파티션으로 그룹화 한다. 때문에 공간 파티셔닝은 클러스터 간의 데이터 셔플을 줄이고, 파티션 상에서 불필요한 데이터 계산을 생략하여 조인 질의 시 질의 속도를 높인다. 또한 각 파티션들 간에 메모리 공간과 공간 계산의 부하 균형을 유지하도록 파티셔닝 해준다. SRDD는 대규모의 분산 데이터 세트에서 모든 SRDD를 조사하고, 공간 분포를 구하는 파티셔닝 작업에 대량의 시간이 소요된다. 때문에 세도나 에서는 각 공간 분포를 탐지하여 낮은 오버헤드로 파티셔닝 작업을 가능하게 하는 기법을 사용한다. 이 기법은 총 3단계로 구성된다.

○ 1 단계 : 글로벌 공간 그리드 파일 구축

스파크 마스터 노드가 각 SRDD 파티션에서 데이터를 수집하고, SRDD의 데이터 분포를 반영한 작은 SRDD의 서브 세트을 생성한다. 이 서브 세트는 부하 균형을 갖는 여러 파티션들로 분할 후, 모든 SRDD에 이 서브 세트의 경계를 적용하여, 공간 근접성을 반영하고 부하 균형을 이루는 새로운 SRDD를 생성한다. 세도나는 에서 제안한 전체 SRDD의 1%만을 샘플링하여 아래와 같은 분할 기법을 적용한다.

- **Uniform grid** : 모든 2차원 공간은 같은 크기의 그리드 셀로 분할하고, 이들 셀들을 전체 SRDD에 적용한다. 이 파티션 방식은 부하균형(LoadBalance)된 그리드를 생성하지 못하므로 골고루 분포된 데이터에 적합하다.

- **R-Tree / Quad-Tree / KDB-Tree** : 이 방식은 공간 트리 인덱스 구조인 R-Tree, Quad-Tree, KDB-Tree의 정의를 활용한다. 각 트리는 같은 수의 자식 노드들을 갖는데, 세도나는 이 트리 중 자식 노드가 없는 노드의 경계를 수집하여 그리드 파일을 생성한다.

○ 2 단계 : 각 객체에 그리드 셀 ID 할당

세도나는 글로벌 그리드 파일 생성한 후에 각 객체가 어느 그리드에 속하는지를 파악하여, 그리드 셀 ID에 따라 분류하여 공간 RDD를 파티션 한다. 즉, 생성된 그리드 파일들을 공간 RDD의 각 파티션에 보내고, 이를 수신한 각 파티션은 그리드 파일을 참조하여 각 객체들을 조사하여 분류하고, <Key, Value> 형태인 <grid ID, object> 으로 새로운 SRDD에 저장한다.

- 여러 그리드에 걸쳐 있는 객체들은 여러 SRDD에 중복되어 저장된다. 세도나의질의 처리 계층에서 중복된 객체를 제거하여 질의의 정확도를 보장한다.

○ 3 단계 : 클러스터에 SRDD를 재 파티션

그리드 셀 ID를 키 값으로 갖는 SRDD에 대하여 키 값을 기준으로 SRDD를 다시 파티션하여 같은 그리드 셀 ID를 갖는 객체들은 같은 파티션으로 그룹화 시킨다. 이 과정은 클러스터의 노드들 간에 공간 객체들을 할당해야 하므로 많은 셔플이 발생한다.

SRDD 인덱싱

R-Tree나 Quad-Tree와 같은 인덱싱 기법들은 인접한 공간 객체들을 트리의 상위 계층의 사각형에 결합하여 표현함으로써 공간 질의를 신속하게 처리한다. 상위 계층의 사각형에 교차(intersect)하지 않는 질의들은 하위 계층의 객체들에서도 교차하지 않는다. 많은 공간 분석 알고리즘들을 수렴할 때까지 같은 공간 RDD에 여러 번 질의를 하기 때문에 세도나는 사용자가 공간 인덱스를 구축하고, 구축된 인덱스를 캐싱, 저장하여 여러 번 재사용한다. 그러나 트리를 사용한 공간 인덱스는 15% 이상의 부가적인 저장 용량 오버헤드를 유발하기 때문에 단일 머신에서 전체 데이터 세트에 구축할 수는 없다. 인덱스 종류는 다음과 같다.

○ 로컬 인덱스 구축 (Build local indexes)

세도나는 사용자가 공간 인덱스 사용을 선택하면 단일 글로벌 인덱스 외에 로컬 공간 인덱스를 구축한다. 즉, 각 RDD 파티션은 공간 인덱스(R-Tree 또는 Quad-Tree)를 생성한다. 이 인덱스 트리는 해당 파티션 내의 공간 객체들 만을 인덱스 한다.

◆ 세도나의 인덱스는 각 파티션 내의 공간 객체들이 공간 인덱스에 직접 저장하는 클러스터 인덱스 (clustered index)를 사용한다. 클러스터 인덱스는 공간 객체들을 바로 리턴하기 때문에 객체의 포인터를 통해 가져오는 입출력 동작이 생략된다.

○ 로컬 인덱스 질의 (Query local indexes)

사용량이 잦은 질의는 병렬로 처리되는 많은 작은 태스크들로 나뉘어진다. 세도나는 특정 파티션에 로컬 인덱스가 존재하면 공간 계산에 이 인덱스를 활용한다. 한번 구축된 인덱스는 계속 재사용되므로 전체 실행 시간을 줄여서, 인덱스 구축 시 소요되는 시간을 상쇄한다.

◆ 공간 인덱스는 객체의 실제 모양 대신 MBR(Minimum Bound Rectangle)를 사용하여 공간 객체들을 조직화하기 때문에 공간 인덱스를 사용하는 모든 질의들은 필터와 정제모델 을 수행해야 한다.

- 필터 단계에서는 질의객체(MBR)와 교차하는 후보 객체들(MBR)을 찾는다.
- 정제 단계에서는 후보 객체와 질의 객체 사이의 공간 관계를 조사하여 요구된 관계를 실제로 만족하는 객체들을 반환 한다.

○ 로컬 인덱스 저장 (Persist local indexes)

사용자는 인덱스된 공간 RDD를 아래와 같이 저장하여 구축된 로컬 인덱스를 재사용 한다.

- ○ IndexedSpatialRDD.cache()를 호출하여 메모리에 캐싱
- ○ IndexedSpatialRDD.saveAsObjectFile(HDFS/S3_PATH)를 호출하여 디스크에 저장

위 두 방식은 같은 알고리즘을 사용하며, 알고리즘은 아래와 같다.

(1) 각 RDD의 파티션을 병렬로 위치시킨다.

(2) SRDD 직렬화기를 호출하여 각 파티션의 로컬 인덱스를 바이트 배열 (파티션 당 하나의 바이트 배열)로 직렬화 한다.

(3) 각 파티션의 생성된 바이트 배열을 메모리 또는 디스크에 저장 한다.

(4) 스파크가 해당 메모리 공간을 관리하고 원본 RDD 파티션 정보로 복구하기 위해 병렬로 바이트 배열를 역직렬화 하기 때문에 사용자는 캐싱된 인덱스 SRDD 이름을 직접 사용할 수 있다. 디스크 상의 인덱스된 SRDD는 IndexedSpatialRDD.readFromObjectFile(HDFS/S3_PATH)를 호출하여 명시적으로 스파크로 읽어 들인다. 스파크는 병렬로 분산 파일 파티션들을 읽어 들여 각 파티션의 바이트 배열을 로컬 인덱스로 역 직렬화 한다.

3.1.2 지리정보 빅데이터 분석 시스템 제안

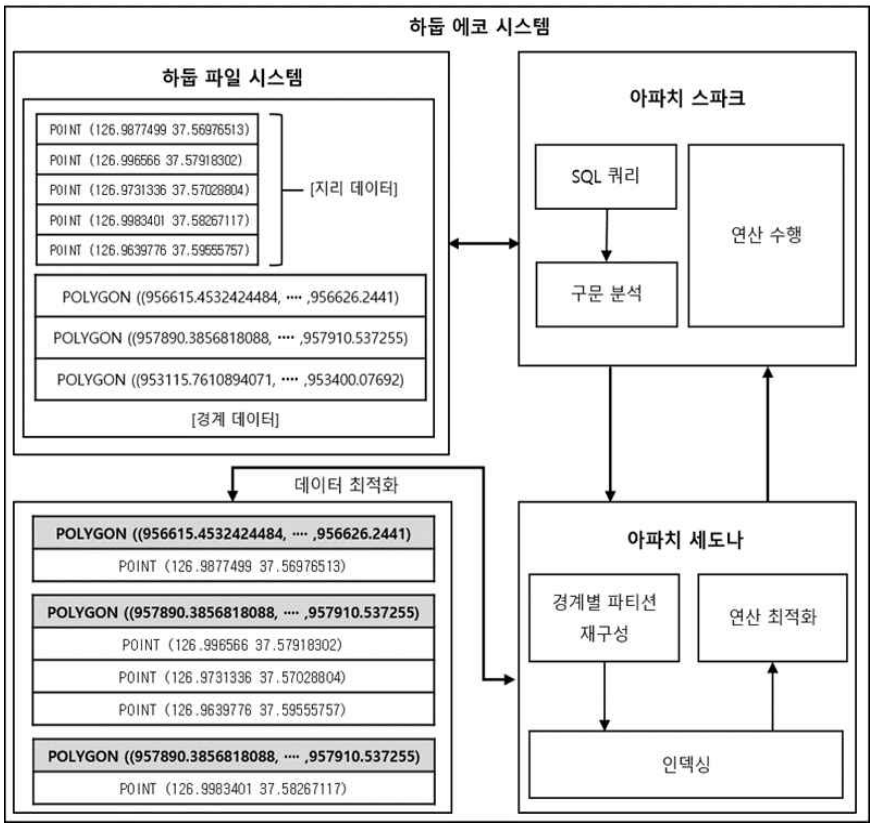
제안하는 시스템은 크게 3가지 부분으로 구성된다. 먼저, 하둡 파일 시스템(HDFS)에 지리정보 데이터를 적재한다. 하둡 파일 시스템은 대용량 파일을 분산된 클러스터에 저장하여 범용 하드웨어에서 실행되도록 설계되었기 때문에 빅데이터를 처리하는데 적합하다.

다음으로 아파치 스파크에서 지리정보 데이터를 필요로 하는 SQL 쿼리문을 실행하게 된다. 스파크로부터 데이터를 요청 받은 하둡 파일 시스템은 스파크로 데이터를 전송해주며, 이를 받은 스파크는 실행된 쿼리문에 대한 구문을 분석하여 공간데이터 연산을 아파치 세도나로 전송해준다. 스파크는 인메모리 기반의 고성능 범용 클러스터 컴퓨팅 시스템으로, SQL과 구조화된 데이터를 처리하기에 적합하기 때문에 SQL 쿼리문을 연산하는 데 용이하다.

마지막으로 세도나는 스파크로 부터 받은 지리정보 데이터에 대하여 파티션을 재구성하게 된다. 파티션 재구성의 경우, 지리정보 데이터가 가지고 있는 경계 데이터를 기준으로 해당 데이터 범위 안에 포함된 지리 데이터를 각각 인덱싱하여 데이터 최적화 과정을 거치게 된다. 인덱싱의 경우 공간 검색의 효율성을 높이기 위하여 각 RDD마다 인덱스를 구축하는 역할이다.

그 후 세도나는 지리정보 데이터에 관한 연산을 최적화 하여 스파크로 전송해주며, 스파크는 이를 받아 최종적으로 연산을 수행하게 된다. 세도나는 원시 데이터를 공간 정보 객체로 변환하여 공간 인덱스를 구축하는 시스템으로, 공간 데이터에 대한 연산을 지원하지 않아 이를 위한 작업을 직접 프로그래밍 해야

하는 스파크의 단점을 보완해준다. [그림 6]은 위에서 제안하는 3가지의 시스템에 대한 처리 과정을 표현한 그림이다.

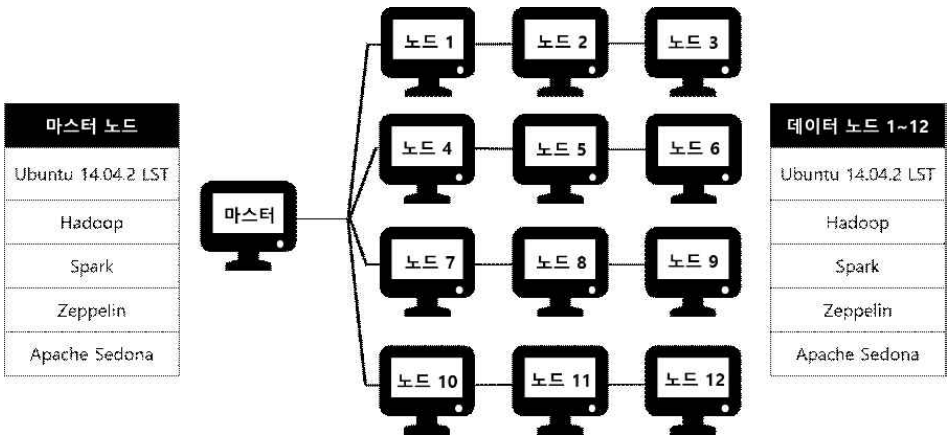


[그림 6] 지리정보 빅데이터 분석 시스템 처리 과정

3.2 지리정보 빅데이터 분석 시스템 환경 구축

아파치 스파크 클러스터를 사용한 지리정보 빅데이터 분석 시스템 환경구축을 하는데 총 13대의 서버가 사용되었다. 마스터인 하나의 네임 노드 1대 와 워커인 12대의 데이터 노드로 구성하였으며 모든 노드의 운영체제는 Ubuntu 14.04.2 LTS 기반으로 하였다. 실험 및 비교를 위해 분산 파일 시스템과 맵리듀스를 위해 하둡을 설치하였으며, 고속의 병렬 처리와 시각화를 위해 스파크와 제플린을 설치하였고 지리 정보 분석을 위해 세도나를 설치하였다.

마스터 노드가 동작 중 멈추게 되었을 경우 또 다른 노드가 새로운 마스터 노드가 되며, 각각의 노드는 여러 개의 샤드(Shard)로 데이터를 분산하여 저장하고, 샤드는 복사본을 갖게 된다. 또한, 노드가 정지하게 될 경우 정지된 노드의 데이터를 다른 노드에 복사한 후 자동으로 정렬된다. 본 논문에서 제안하는 시스템의 노드 구성도는 [그림 7]과 같으며 시스템 사양 및 정보는 아래 [표 1]와 같다.



[그림 7] 노드 구성

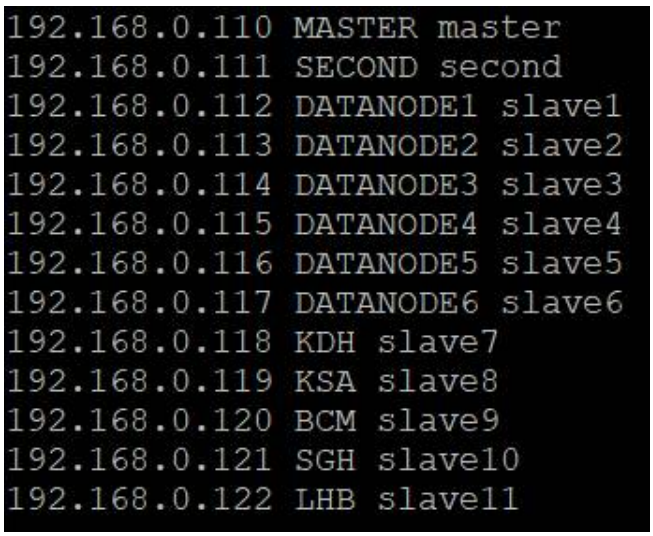
	CPU	RAM	HDD	OS	Tool
마스터노드	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 1	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 2	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 3	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 4	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 5	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 6	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 7	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 8	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 9	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 10	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 11	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1
데이터노드 12	Intel(R) Core(TM) i3-4150 CPU @ 3.50GHz	8G	2TB	Ubuntu 14.04.2 LTS	Hadoop 2.7.7 Spark 2.3.3 Zeppelin 0.8.0 Apache Sedona 1.3.1

[표 1] 노드 구성 사양정보 및 툴

3.2.1 지리정보 빅데이터 분석 시스템 클러스터 환경구축

호스트 파일 설정

도메인 네임 서버에서 주소 정보를 제공받지 않고도 서버의 위치를 찾기 위해서는 [그림8]와 같이 각 노드의 /etc/hosts 파일에 호스트 이름에 대응하는 IP 주소를 저장해 주어야 한다.



[그림 8] 호스트 파일 설정

환경변수 추가

우분투 apt 저장소를 통해서 설치한 패키지들은 보통 자동으로 경로에 추가되어 터미널 어디서든 명령어를 실행할 수 있다. 하지만 직접 설치한 프로그램들은 매번 직접 디렉토리 경로를 찾아 쉘 스크립트 혹은 프로그램을 직접 실행시켜야 하는 번거로움이 있기 때문에 .bashrc파일에 환경변수를 추가하여 이러한 번거로움을 해결하였다.

```
export HADOOP_PREFIX=$HOME/hadoop-2.7.7
export HADOOP_HOME=$HOME/hadoop-2.7.7
export PATH=$PATH:$HADOOP_PREFIX/bin
export PATH=$PATH:$HADOOP_PREFIX/sbin
export HADOOP_MAPRED_HOME=${HADOOP_PREFIX}
export HADOOP_COMMON_HOME=${HADOOP_PREFIX}
export HADOOP_HDFS_HOME=${HADOOP_PREFIX}
export YARN_HOME=${HADPPT_PREFIX}
export HADOOP_YARN_HOME=${HADOOP_PREFIX}
export HADOOP_CONF_DIR=${HADOOP_PREFIX}/etc/hadoop
export YARN_CONF_DIR=${HADOOP_PREFIX}/etc/hadoop

#Native Path
export HADOOP_COMMON_LIB_NATIVE_DIR=${YARN_HOME}/lib/native
export HADOOP_OPTS="-Djava.library.path=$YARN_HOME/lib"

#SPark Path
export   SPARK_HOME=$HOME/spark-2.3.3-bin-hadoop2.7
export LD_LIBRARY_PATH=$HADOOP_HOME/lib/native
```

데이터 노드 설정

데이터 노드를 설정해주기 위해 /Hadoop-2.7.7/etc/hadoop/slaves 경로에 위치한 파일을 열어 모든 노드들의 도메인을 입력해준다.

```
MASTER
SECOND
DATANODE1
DATANODE2
DATANODE3
DATANODE4
DATANODE5
DATANODE6
KDH
KSA
BCM
SGH
LHB
```

마스터 노드 및 URL 설정

/hadoop-2.7.7/etc/hadoop/core-site.xml 경로에 위치한 파일에서 클러스터 내 마스터 노드에서 실행되는 하둡 데몬에 관한 설정을 해준다. 기본 파일 시스템 이름을 설정해주면 된다.

```
<configuration>
        <property>
                <name>fs.default.name</name>
                <value>hdfs://MASTER:9000</value>
        </property>
</configuration>
```

마스터 노드 및 데이터 노드 경로 지정

/hadoop-2.7.7/etc/hadoop/hdfs-site.xml 경로에 위치한 파일에서 네임스페이스(namespace)와 트랜잭션 로그를 저장 할 마스터 노드와 데이터 노드의 저장 경로를 지정하고, 데이터 복제 개수를 설정해준다. 이는 마스터 노드와 데이터노드 그리고 하둡 파일 시스템 데몬을 위한 환경설정이다.

```
<configuration>
        <property>
                <name>dfs.replication</name>
                <value>3</value>
        </property>
        <property>
                <name>dfs.name.dir</name>
                <value>/home/hadoop/hadoop-2.7.7/hdfs/name</value>
        </property>
        <property>
                <name>dfs.data.dir</name>
                <value>/home/hadoop/hadoop-2.7.7/hdfs/data</value>
        </property>
        <property>
                <name>dfs.http.address</name>
                <value>MASTER:50070</value>
        </property>
        <property>
                <name>dfs.permissions</name>
                <value>false</value>
        </property>
        <property>
                <name>dfs.namenode.secondary.http-address</name>
                <value>SECOND:50090</value>
        </property>
        <property>
                <name>dfs.namenode.secondary.https-address</name>
                <value>SECOND:50091</value>
        </property>
</configuration>
```

얀(yarn) 자원 관리자 및 노드 관리자 환경설정

/hadoop-2.7.7/etc/hadoop/yarn-site.xml 경로에 위치한 파일에서 리소스 매니저 웹 UI 주소, 노드매니저에서 중간단계 파일 및 로그를 저장할 경로를 정의해 준다.

```
<configuration>

<!-- Site    specific YARN configuration properties -->
        <property>
                <name>yarn.nodemanager.aux-services</name>
                <value>mapreduce_shuffle</value>
        </property>
        <property>

<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
                <value>org.apache.hadoop.mapred.ShuffleHandler</value>
        </property>
        <property>

<name>yarn.resourcemanager.resource-tracker.address</name>
                <value>master:8025</value>
        </property>
        <property>
                <name>yarn.resourcemanager.scheduler.address</name>
                <value>master:8030</value>
        </property>
        <property>
                <name>yarn.resourcemanager.address</name>
                <value>master:8050</value>
        </property>
        <property>
                <name>yarn.resourcemanager.webapp.address</name>
                <value>master:8088</value>
        </property>
        <property>
                <name>yarn.nodemanager.resource.memory-mb</name>
                <value>2048</value>
        </property>
        <property>
                <name>yarn.nodemanager.resource.cpu-vcores</name>
                <value>1</value>
```

```
        </property>
        <property>
                <name>yarn.nodemanager.pmem-check-enabled</name>
                <value>false</value>
        </property>
          <property>
                <name>yarn.nodemanager.vmem-check-enabled</name>
                <value>false</value>
        </property>
</configuration>
```

맵 리듀스 설정

/hadoop-2.7.7/etc/hadoop/mapred-site.xml.template 경로에 위치한 파일에서 맵 리듀스의 JobTracker, TaskTracker 데몬을 위한 설정을 해준다.

```
<configuration>
        <property>
                <name>mapreduce.framework.name</name>
                <value>yarn</value>
        </property>
</configuration>
```

마스터 노드 .bashrc 파일 배포

마스터 노드의 .bashrc 파일을 모든 데이터 노드의 하둡 폴더에 배포해준다.

```
$ scp    /home/bigdata/.bashrc 데이터노드:~
$ scp -r    /home/bigdata/hadoop-2.7.7 데이터노드:~
```

스파크 기존 파일 백업 및 설정 변경

스파크의 기존 파일을 복사하여 백업을 한 후, 복사한 파일을 스파크 설정 파일 변경에 사용한다.

```
// 파일 복사
$ cp slaves.template    slaves
$ cp spark-defaults.conf.template    spark-defaults.conf
$ cp spark-env.sh.template    spark-env.sh
```

spark-defaults.conf 파일 수정

/spark-2.3.3-bin-hadoop2.7/conf/spark-defaults.conf 경로의 설정 파일 내용 변경

```
spark.master                    spark://master:7077
spark.yarn.jars                 hdfs://master:9000/jar/spark-jars/*.jar
spark.driver.extraClassPath         /home/bigdata/hbase-2.2.0/lib/*.jar
spark.executor.extraClassPath     /home/bigdata/hbase-2.2.0/lib/*.jar
```

conf/spark-env.sh 파일 수정

/spark-2.3.3-bin-hadoop2.7/conf/spark-env.sh 경로의 설정 파일 내용 변경

```
SPARK_WORKER_MEMORY    = 2g
export HADOOP_CONF_DIR=${HADOOP_DIR}/etc/hadoop
```

스파크 jar 파일 업데이트

마스터 노드 에서 얀 모드 실행을 위한 작업이다.

```
$ hdfs dfs  - mkdir /jar
$ hdfs dfs  - mkdri /jar/spark-jars
$ hdfs dfs  - put $SPARK_HOME/jars/* /jar/spark-jars/
```

마스터 노드의 스파크 폴더 배포

마스터 노드의 스파크 폴더를 모든 데이터 노드에 배포해준다.

```
$ scp    -r spark-2.3.3-bin-hadoop2.7 데이터노드:~/
```

3.2.2 아파치 세도나(Apache Sedona) 환경 구축

세도나를 사용하기 위해서는 하둡 파일시스템의 /jar/spark-jars 경로에 세도나 jar 파일을 추가해 주어야 한다. 세도나는 총 세 가지의 종류로 나뉘어져 있으며 이는 세도나, 세도나-sql, 세도나-viz 로 구성되어져 있다. 세도나의 핵심 기능인 지리 정보 분석 기능을 사용하기 위해서는 위의 세 가지 모두를 사용하여야 하며, 특히 세도나-sql의 경우 지리 정보 분석을 위한 별도의 함수가 제공되기 때문에 가장 중요한 요소이다.

세도나 jar 파일은 메이븐(Maven) 공식 홈페이지에서 쉽게 다운받을 수 있다. 다운 받을 jar 파일의 경우 사용하고 있는 아파치 계열의 응용 소프트웨어 버전과 맞춰서 다운을 받아 주어야 한다. 그래야 세도나의 기능을 올바르게 사용할 수 있다.

다운받은 jar 파일은 웹 서버로부터 콘텐츠를 가져오는 wget 기능을 사용하여 다운로드 받았다. 아래 [그림 9]은 옮겨준 jar 파일을 하둡 파일시스템 웹을 통해 보여주는 그림이다. jar 파일은 1.3.1 버전으로 통일시켜 다운받아 준 모습을 확인할 수 있다.

-rw-r--r--	hadoop	supergroup	11.42 MB	2020. 10. 26. 오후 9:46:03	3	128 MB	geospark-1.3.1.jar
-rw-r--r--	hadoop	supergroup	275.84 KB	2020. 11. 2. 오후 5:31:33	3	128 MB	geospark-sql_2.3-1.3.1.jar
-rw-r--r--	hadoop	supergroup	13.79 MB	2020. 10. 26. 오후 8:45:10	3	128 MB	geospark-viz_2.3-1.3.1.jar

[그림 9] 하둡 파일 시스템으로 옮겨준 세도나 jar 파일 목록

제 4 장 공간 연산 성능 평가

본 장 에서는 제안한 시스템인 스파크 클러스터를 사용한 시스템의 성능을 평가하기 위한 실험환경을 구축하여 성능평가를 하였다. 스파크 클러스터를 이용한 지리정보 빅데이터 분석 시스템은 약 2,400만개의 데이터를 하둡 파일 시스템에 적재시켜 스파크를 통해 분석하였으며, 시각화 도구인 제플린을 이용해 분석 처리 시간을 확인하고 비교하는 실험을 진행하였다.

1대의 마스터 노드와 12대의 작업 노드들을 기준으로 작업 노드의 개수를 1, 4, 8, 그리고 12개로 점차 늘려가며 실험 하였다. 즉, 지리정보 빅데이터를 분석하는데 있어 몇 대의 클러스터를 구성해야 신속하고 빠른 결과를 도출해 낼 수 있는가에 대한 실험을 하였다. 이를 위해 대표적인 공간 연산, 공간 범위 질의(Spatial range query), 공간 KNN 질의(spatial K-Nearest Neighbors query), 공간 조인 질의 (Spatial join query) 를 적용하여 성능을 평가 하였다. 이 실험 결과를 토대로 지리정보 빅데이터 분석 시스템에 대한 성능 평가를 진행하였다.

4.1 공간 연산 환경 구성

실험을 위한 환경 구성을 위해 스파크 클러스터링 된 13대의 노드와 파일 관리를 위한 1대의 마스터 노드로 구성하였다. 총 13대의 노드 중 1대는 마스터 노드로 분석 및 통계 서버의 역할을 하며 나머지 12대는 데이터 노드이다. 시스템 구성은 다음과 같이 구성하였다.

○ 마스터노드 스파크 클러스터 마스터 구성

- 하둡, 스파크, 제플린, 세도나

○ 데이터노드 스파크 클러스터 구성

- 하둡, 스파크, 제플린, 세도나

○ 노드 시스템 사양

- CPU i3-4150 3.50GHz
- RAM : 8GB
- HDD : 2TB

4.2 공간 연산 진행 방법

4.2.1 전처리

각각의 데이터가 가지고 있는 인덱스가 다르기 때문에 이를 확인한 후 데이터가 가지고 있는 인덱스 형식에 맞춰 스키마를 생성하였고, 이를 데이터프레임 또는 SRDD에 저장하여 주었다. 저장 후, SQL문 사용을 위해 데이터프레임 및 SRDD를 뷰(View)로 등록해 주었다 [그림 10]. 데이터는 버스 승객 승하차 데이터, 시간대별 버스 승객 승하차 데이터, 버스 정류장 좌표 데이터, 서울시 행정구역 구 단위 데이터로 나뉘어져 있다.

```
println("1월")
val Bus_Schema = new StructType(Array(
new StructField("Use_Date", StringType, true),
new StructField("bus_route_no", StringType, true),
new StructField("bus_route_nm", StringType, true),
new StructField("Bus_ARS", IntegerType, true),
new StructField("Station_name", StringType, true),
new StructField("Riding_all_user", IntegerType, true),
new StructField("Quit_all_user", IntegerType, true),
new StructField("work_dt", IntegerType, true)
))

val Bus_1DF = spark.read.format("csv").schema(Bus_Schema).load("/sparkdata/join/bus/Bus1.csv").toDF("Use_Date", "bus_route_no", "bus_route_nm", "Bus_ARS"
    ,"Station_name","Riding_all_user","Quit_all_user","work_dt").filter("Use_Date is not null")

//Bus_1DF.show(10,0)
//Bus_1DF.count()
Bus_1DF.createOrReplaceTempView("Bus_1DF")
```

```
var spatialRDD = new SpatialRDD[Geometry]
spatialRDD = ShapefileReader.readToGeometryRDD(sc, "/sparkdata/join/SIG/SIG4.*")

var rawSpatialDf = Adapter.toDf(spatialRDD,spark)

rawSpatialDf.createOrReplaceTempView("rawSpatialDf")
```

[그림 10] 스키마 생성, 데이터 프레임, SRDD 및 뷰 등록

시간대별 버스 승객 승하차 데이터의 경우 컴퓨팅오버헤드를 줄이기 위해 각각 캐시(Cache)에 등록해주었다. 이는 RDD를 메모리에 영구적으로 저장해주기 때문에 이후에 해당 결과가 필요 하게 될 경우 결과를 바로 불러와서 사용이 가능하다. [그림 11]는 시간대별 버스 승하차 데이터를 캐시로 등록해준 모습이다.

```
//9월
val Bus_Time_9mDF = spark.read.format("csv").schema(Bus_Time_Schema).load("/sparkdata/join/bus_time/bt9.csv").toDF("Use_Date", "Bus_ARS", "Station_name", "_00h_Riding",
    "_00h_Quit", "_01h_Riding",
"_01h_Quit", "_02h_Riding", "_02h_Quit", "_03h_Riding", "_03h_Quit", "_04h_Riding", "_04h_Quit", "_05h_Riding", "_05h_Quit", "_06h_Riding", "_06h_Quit", "_07h_Riding",
    "_07h_Quit", "_08h_Riding",
"_08h_Quit", "_09h_Riding", "_09h_Quit", "_10h_Riding", "_10h_Quit", "_11h_Riding", "_11h_Quit", "_12h_Riding", "_12h_Quit", "_13h_Riding", "_13h_Quit", "_14h_Riding",
    "_14h_Quit", "_15h_Riding",
"_15h_Quit", "_16h_Riding", "_16h_Quit", "_17h_Riding", "_17h_Quit", "_18h_Riding", "_18h_Quit", "_19h_Riding", "_19h_Quit", "_20h_Riding", "_20h_Quit", "_21h_Riding",
    "_21h_Quit", "_22h_Riding",
"_22h_Quit", "_23h_Riding", "_23h_Quit", "_24h_Riding", "_24h_Quit")
Bus_Time_9mDF.cache()
Bus_Time_9mDF.createOrReplaceTempView("BT9")
```

[그림 11] 시간대별 버스 승하차 데이터 캐시 등록

또한, 시간대별 버스 승객 승하차 데이터는 각 시간대별로 승하차 승객에 대한 값을 뽑아서 사용해야 한다. 이 데이터의 경우 0시부터 23시 까지 총 24시간을 9개월 분량의 승하차 데이터로 나누어서 사용해야 했는데, 이는 시간적인 부분에서 낭비가 크다고 생각하였다.

이러한 문제를 해결하고자 데이터를 전처리하기 위한 함수를 만들어 적용시켜 주었다. 해당 함수는 시간대별 버스 승객 승하차 데이터를 월별로 나누어 각각 승객이 승차를 했는지 하차를 했는지 구별해주는 함수이며, 마찬가지로 캐시에 등록하여 사용하였다. [그림 12]은 함수에 대한 코드이고 [그림 13]은 함수 출력 화면이다. [그림 13]의 경우 제플린 차트를 사용하여 결과 값을 보기 좋게 시각화 하였다.

```
//시간대를 저장하는 List
val hour = List("00","01","02","03","04","05","06","07","08","09","10","11","12","13","14","15","16","17","18","19","20","21","22","23")
//print(hour.length)

//List 에 DataFrame type 저장
var Bus_list = new ListBuffer[DataFrame]()

//1월부터 9월까지의 데이터 첫번째 포문
for(k <-1 to 9){
    //00 시부터 23 시까지 총 24번 루프
    for (i <- 0 until 23){
        //pp는 타는시간 00h_Riding
        //oo는 내리는시간 00h_Quit
        val pp = s"_${hour(i)}h_Riding"
        val oo = s"_${hour(i)}h_Quit"
        //println(pp)
        //정류장별 월별 탄 인원, 내린 인원을 데이터 프레임으로 저장
        val Riding = spark.sql(s"SELECT s.Station_name,s.Bus_ARS,s.Use_Date,t.* FROM (SELECT MAX(${pp}) AS MAX_VALUE FROM BT1) t, BT${k} s WHERE s.${pp} = t.MAX_VALUE").toDF
            ().withColumn("Time",lit(s"${hour(i)}")).withColumn("inout",lit("Riding")).withColumn("month",lit(s"month${k}"))
        val Quit = spark.sql(s"SELECT s.Station_name,s.Bus_ARS,s.Use_Date,t.* FROM (SELECT MAX(${oo}) AS MAX_VALUE FROM BT1) t, BT${k} s WHERE s.${oo} = t.MAX_VALUE").toDF
            ().withColumn("Time",lit(s"${hour(i)}")).withColumn("inout",lit("Quit")).withColumn("month",lit(s"month${k}"))
        //데이터 프레임을 List에 추가
        Bus_list += Riding
        Bus_list += Quit
    }
}
//val list_b = Bus_list.toList()

print(Bus_list.length)

//List의 DataFrame을 하나씩 Union
var total = Bus_list(0)
for(i <- 1 to Bus_list.length-1){
    total = total.union(Bus_list(i))
}
//total.show()
total.cache()
```

[그림 12] 시간대별 버스 승객 승하차 전처리 함수

Station_name	Bus_ARS	Use_Date	MAX_VALUE	Time	inout	month
신림사거리	21153	202001	2684	00	Riding	month1
쌍문동금호1차삼익아파트	10215	202001	1253	00	Quit	month1
의정부역.흥선지하도	61009	202001	1080	01	Riding	month1
수유역.강북구청	9004	202001	833	01	Quit	month1
연신내역.연서시장	12018	202001	849	02	Riding	month1
연신내역.연서시장	12017	202001	555	02	Quit	month1
강남역.역삼세무서	23287	202001	830	03	Riding	month1
종로2가	1014	202001	568	03	Quit	month1
잠실역.롯데월드	24138	202001	1722	04	Riding	month1
길음뉴타운	8003	202001	1171	04	Quit	month1
삼익아파트앞	18220	202001	1824	05	Riding	month1
신대방역	21225	202001	2520	05	Quit	month1
가산디지털단지역	18503	202001	4670	06	Riding	month1

[그림 13] 승하차 전처리 함수 출력 화면

다음으로, 지리정보를 처리하는 데 있어 가장 중요한 좌표 데이터이다. 전처리 작업을 거치기 전에는 위도(latitude) 와 경도(longitude) 값이 각각 두 개의 컬럼으로 나뉘어져 있었다. 하지만 세도나를 사용하여 지리공간 분석을 하기 위해 이 두 좌표 값을 하나의 좌표로 묶어준다. [그림 14]는 그 예로 앞서 말한 두 컬럼을 합쳐 포인트(Point) 좌표를 생성한 데이터 이다.

Station_name	Bus_ARS	Longitude	Latitude	geom	Gu
종로2가사거리	1001	126.98775	37.569767	POINT (126.9877499 37.56976513)	종로구
창경궁.서울대학교병원	1002	126.99657	37.57918	POINT (126.996566 37.57918302)	종로구
명륜3가.성대입구	1003	126.99834	37.582672	POINT (126.9983401 37.58267117)	종로구
종로2가.삼일교	1004	126.98761	37.56858	POINT (126.9876131 37.56857927)	종로구
혜화동로터리.여운형활동터	1005	127.00175	37.586243	POINT (127.001744 37.586243)	종로구
경기상고	1101	126.96936	37.58755	POINT (126.9693635 37.58755022)	종로구
신교동	1102	126.97027	37.583267	POINT (126.9702654	종로구

[그림 14] 버스 정류장 좌표 데이터

마지막으로 경계 좌표 데이터(Polygon)가 있다. 위에서 말했던 데이터가 각 장소에 대한 위치를 나타내주는 좌표 데이터였다면, 경계 좌표 데이터는 이 좌표 값들이 속해있는 장소. 즉, 행정구역에 관한 좌표 데이터라고 할 수 있다. 행정구역은 크게 4가지로 분류된다. 단위가 제일 작은 구역 순서로 리, 읍면동, 시군구, 시도 이렇게 4 가지로 나뉜다. 본 연구자는 시군구를 기준으로 나눈 경계 좌표 데이터를 사용하였다. 만약 서울시를 시군구로 나누어 광진구와 종로구로 크게 봤을 때, '광진청소년수련관' 이라는 정류장은 광진구에 속해있고 '종로1가' 라는 정류장은 종로구에 속해있듯 포인트 좌표가 어떠한 경계 범위 안에 속해있을 때, 그 범위를 나타내는 좌표 데이터를 나타낸다. [그림 15]는 광진구(경계 좌표 데이터)에 속해있는 포인트 좌표(지리 데이터 : 위도, 경도)를 나타내는 폴리곤 형식의 데이터이다.

[그림 15] 경계 좌표 데이터

4.2.2 공간 연산 적용

공간 범위 질의 (Spatial range query)

공간 범위 질의는 특정 영역 내의 공간 객체들을 반환한다. 예를 들어 서울시 광진구에 속한 모든 정류장들을 검색하거나, 사람의 현재 위치에서 1.6km 이내의 모든 장소들을 반환한다. 입력 형식에 따라 포인트, 폴리곤, 질의 윈도우(라인 스트링 등) 등을 입력으로 받아 질의 영역 내의 모든 포인트 또는 폴리곤 데이터들을 반환한다. 단, 입력 질의 범위 안에 있는 모든 공간 객체를 리턴하기 때문에 빠르고 적은 자원만 요구된다. 스파크에서는 좁은 속성만 갖기 때문에 병렬 변환만으로 질의를 수행할 수 있다. 세도나의 공간 SQL에서 공간 범위 질의는 “ST_Contains″ 와 같은 공간 판단식을 사용한다. 설명은 아래와 같다.

- “ST_Contains(A, B)″ 는 A가 B를 포함하면 참을 반환한다.
- “ST_Distance(A, B) d″ 는 A와 B 사이의 거리가 d 보다 작거나 같으면 참을 반환한다.

아래 [그림 16]은 (1) 광진구에서 승객이 승차했던 버스 정류장을 반환하고, (2) 광진구에서부터 1 마일 (1.6km) 거리 이내에서 승차한 승객의 버스 정류장을 반환 한다.

```
var range_1 = spark.sql("SELECT * FROM sum WHERE ST_Contains(Gwangjin-gu, sum.geom)")
var range_2 = spark.sql("SELECT * FROM sum WHERE ST_Distance(Gwangjin-gu, sum.geom) <= 1")
```

[그림 16] 공간 범위 질의를 사용한 SQL

공간 범위 질의 알고리즘

공간 범위 질의 시 질의 윈도우가 클러스터의 각 머신으로 전송된다. 각 SRDD 파티션에 대해 아래 내용을 수행하고 결과 SRDD는 스파크 프로그램의 다음 스테이지로 보내지거나 디스크에 저장된다.

- 인덱스가 존재하면 아래와 같은 필터 세분화 모델을 수행한다.

- 공간 인덱스 색인에 질의 윈도우 MBR(최소 경계 사각형)을 적용하여 후보 결과들을 반환 한다.

- 질의 윈도우와 후보 객체들 간의 실제 모양을 적용하여 이들 간의 관계를 조사하고 검증된 공간 객체들이 결과 SRDD로 반환된다.

- 인덱스가 존재하지 않으면, 세도나는 질의 윈도우를 사용하여 공간 객체들을 필터링하고 검증된 객체들을 수집하여 새로운 결과를 SRDD로 반환한다.

- 셔플이 필요하지 않은 하나의 좁은 종속성만가지고 범위 질의를 처리한다. 이는 서로 다른 질의 윈도우들에 대해 범위 질의를 여러 번 수행하기 때문에 자주 사용 하는 인덱싱된 SRDD를 캐싱하여 접근하므로 아주 빠른 질의 처리가 가능하다.

공간 KNN 질의 (Spatial K nearest neighbors query)

공간 KNN 질의는 중심 포인트, 공간 객체 세트를 입력으로 받아 중심 포인트에서 가장 가까운 K 개의 이웃들을 검색한다. 예를 들어 사람 주변의 가장 가까운 10개의 정류장을 찾는 질의가 있다.

KNN질의를 단순하게 실행하는 방법은 공간 객체들과 질의 지점 사이의 거리를 정렬하여, 상위 K개의 가까운 이웃들을 선택하는 것이다. 그러나 거리 기준 정렬은 대규모 데이터 셔플을 초래하기 때문에 시간이 많이 걸리고 대역폭을 소모하여 피해야 한다.

SQL API 공간 SQL에서는 “ST_Neighbors(A, B, K)”를 사용하여 B에서 A의 가장 가까운 K개의 이웃들을 찾는다. [그림 17]은 종로구에서 가장 가까운 버스 승차 지점 100개를 반환하는 예이다.

```
var range_3 = spark.sql("SELECT ST_Neighnors(Jongno-gu, sum.geom, 100) from sum")
```

[그림 17] 공간 KNN 질의를 사용한 SQL

공간 KNN 질의 알고리즘

공간 KNN 질의를 분산 환경에서 병렬 처리하기 위해 로컬 공간 인덱스를 활용하고 거리를 정렬하는 데이터 셔플의 규모를 줄이도록 변경되었다. 알고리즘은 SRDD와 질의 중심점 (포인트, 폴리곤, 라인 스트링 등), K 값을 입력으로 받아 선택과 정렬 두 단계를 수행한다.

○ 선택 단계

세도나는 각 SRDD 파티션에 대해 중심점과 공간 객체들과 거리를 계산하고 거리에 따라 객체들을 추가 제거하는 로컬 우선순위 큐를 관리한다. 인덱스된 SRDD에서는 빠른 거리 계산을 위해 파티션의 로컬 인덱스에 질의할 수 있다. 앞에서와 마찬가지로 인덱스 색인으로 반환된 결과는 필터 및 정제 모델을 적용하여 검증된다.

○ 정렬 단계

선택 단계에서 SRDD의 각 파티션에서 생성된 중간 단계 SRDD의 모든 K개 의 객체들을 거리 기준으로 올림차순으로 정렬하여 K개의 객체들을 리턴한다. 이는 원본 SRDD의 모든 객체들을 정렬하는 것보다 훨씬 빠르다.

공간 조인 질의 (Spatial join query)

공간 조인 질의는 거리 제약 등과 같은 공간적인 판단식(spatial predicate)을 갖는 두 개 이상의 데이터 세트들을 결합한 질의이다. 예를 들어 지하철 정류장을 포함하고 있는 버스 정류장을 찾는 질의들이 이에 해당한다.

공간 조인 질의는 포인트, 사각형 또는 폴리곤들의 세트와 질의 윈도우 세트를 입력으로 받아서 각 질의 윈도우 세트 내에 있는 모든 포인트와 폴리곤들을 반환한다.

공간 조인은 많은 계산을 하는 연산이기 때문에 스파크에서 큰 데이터 셔플을 유발한다. 세도나의 조인 알고리즘은 공간 인접성에 따라 파티션되고 캐싱된 SRDD를 활용하여 큰 데이터 셔플을 방지할 수 있다. 또한 조인 질의 판단식을 만족하지 않는 파티션들은 생략이 가능하기 때문에 공간 조인 처리를 빠르게 수행할 수 있다.

[그림 18]은 버스 정류장에서 특정 거리 내에 있는 모든 공간 객체 쌍들을 반환하는 질의 이다.

```
var range_4 = spark.sql("SELECT * FROM total a, sum b WHERE ST_Contains(b.geom, a.polygon")
var range_5 = spark.sql("SELECT * FROM total a, sum b WHERE ST_Contains(b.geom, a.polygon") <= 1
```

[그림 18] 공간 조인 질의를 사용한 SQL

공간 조인 질의 알고리즘

공간 조인 질의는 두 개의 공간 RDD A 와 B를 입력으로 받아 아래와 같이 두 가지의 방식으로 나누어진다.

○ 첫 번째 방식

클러스터에서 교차하는 모든 객체를 수집하여 GroupBy 연산을 적용하여 중복된 객체를 제거하는 것이다. 이 방식은 클러스터의 모든 결과에 대해 GroupBy를 적용하므로 많은 데이터 셔플을 유발한다.

○ 두 번째 방식

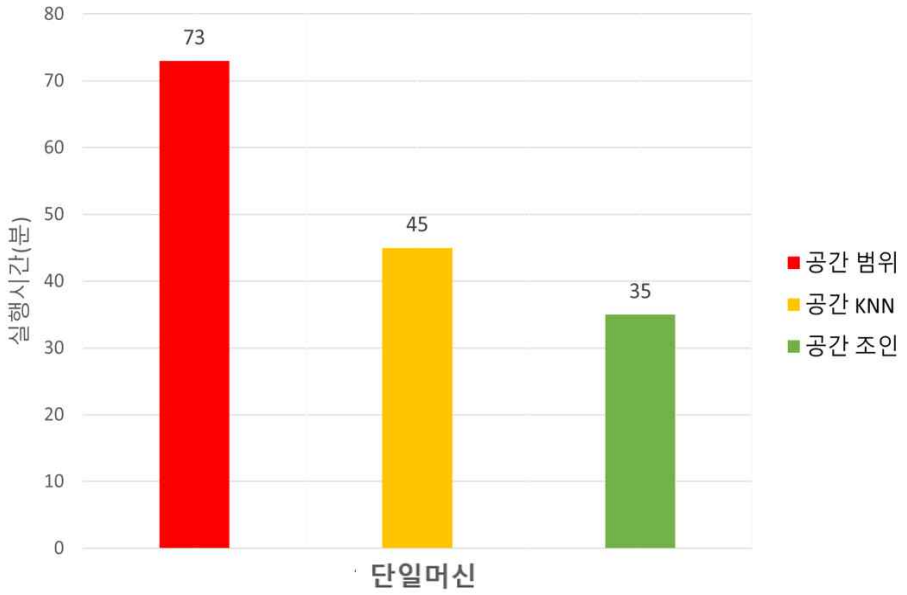
기준점을 적용하여 중복이 발생하는 경우 하나만을 보고한다. 파티션 수준 로컬 조인 시 A, B의 두 객체들 간의 교차를 계산하고 기준점이 해당 그리드에 속한 객체들의 쌍만을 보고한다. 두 교차하는 객체의 기준점은 교차되는 모든 좌표 (X, Y) 중 최대 값의 (X, Y)가 기준점이 된다. 이 기준점 방식은 그리드가 겹치지 않는 Quad-Tree, KDB-Tree 파티션에서만 적용할 수 있다. R-Tree와 같이 그리드가 겹치는 파티션에서는 기준점이 여러 그리드에서 존재할 수 있으므로 첫 번째 방식을 적용해야 한다.

4.3 공간 연산 성능 평가

성능 평가에 사용된 데이터는 위치정보를 포함하는 대중교통(버스) 데이터로, [참고문헌 2,3,4]에서 제공하는 공공데이터이다. 수집된 데이터의 수는 약 2400만 개로, 용량은 1.2GB이다. 데이터는 실제 대중교통 데이터로 사용자들이 이용하면서 누적된 정보이다. 추가적으로 버스 정류장에 대한 지리좌표와 경계좌표 데이터도 있다. 실험에 사용된 분산 시스템은 3.1장 [표1]과 같다. 클러스터 수에 따른 실행 시간을 비교하기 위하여 단일머신 테스트와 클러스터의 수가 4, 8, 그리고 12인 경우의 실험을 실행하였다.

4.3.1 단일머신

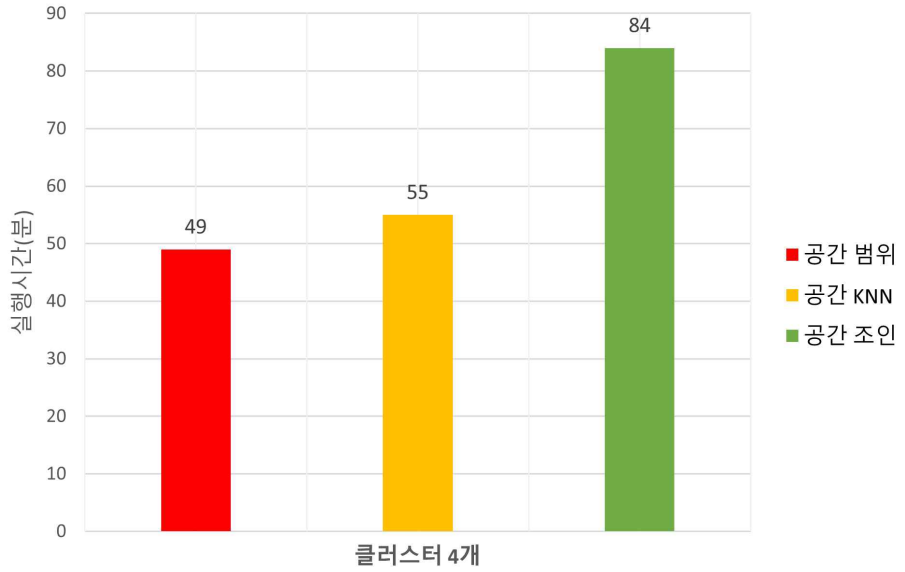
단일 머신 에서의 연산 처리는 세 가지의 공간 질의를 사용하여 진행하였다. 공간 범위 질의, 공간 KNN 질의, 공간 조인 질의로 진행 하였으며, 클러스터 수에 따른 성능 평가를 위해 단일 머신으로 수행 성능을 실험 하였다. [그림 19]는 각각의 공간 질의를 사용하여 수행 성능을 실험한 결과이며, 공간 범위 질의가 공간 KNN 질의와 공간 조인 질의보다 빠름을 알 수 있다.



[그림 19] 단일머신에서의 공간 질의 연산 처리

4.3.2 클러스터

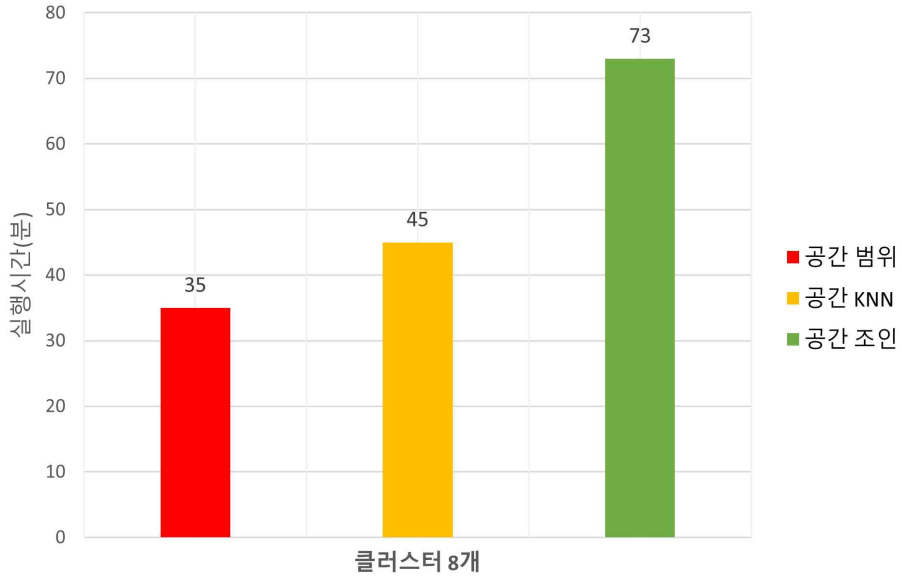
클러스터를 이용한 테스트는 분산처리 컴퓨팅의 특성상 노드가 늘어남에 따라 그 처리 성능도 증가하는 것을 알 수 있는데, 아래의 클러스터를 사용한 실험 결과가 증명 하듯, 3가지 방법의 클러스터 실험 모두 단일 머신 실험에 비하여 월등한 처리 속도를 보여주었다.



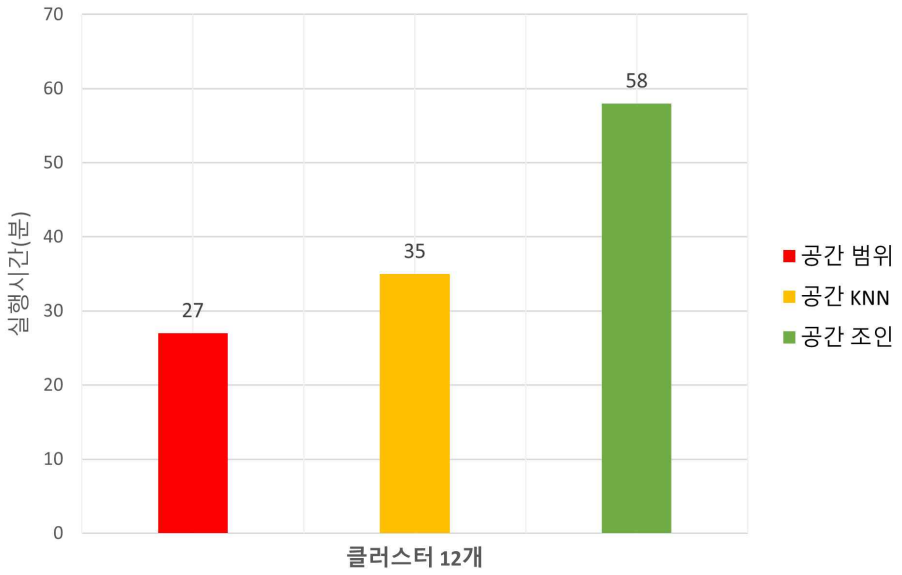
[그림 20] 클러스터 4개를 사용한 공간 질의 연산 처리

그 중에서도 공간 범위 질의가 49분이라는 빠른 속도를 보여주며, 단일머신 실험과 같이 가장 빠른 처리속도를 보인다. 공간 KNN 질의와 공간 조인 질의도 단일머신 실험과 유사하게 약 1.1배 이상의 속도차이를 보이며 데이터를 처리하였다. 단일머신 실험과 같이 [그림 21]과 [그림 22]의 클러스터 실험 에서는 단일머신 실험 증가폭과 달리 3가지 실험 모두 2배 이상 증가하지 않는 모습을 확인

할 수 있다.

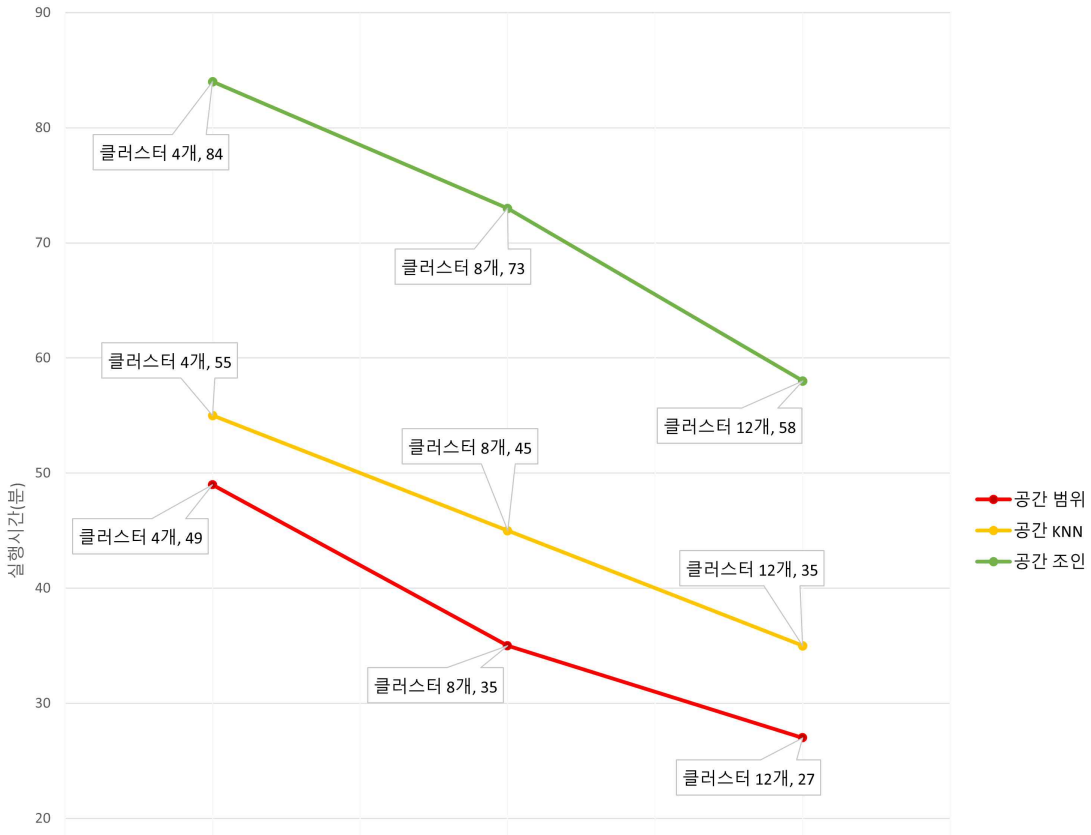


[그림 21] 클러스터 8개를 사용한 공간 질의 연산 처리



[그림 22] 클러스터 12개를 사용한 공간 질의 연산 처리

클러스터 실험을 종합한 결과는 [그림 23]과 같다. 앞서 단위 테스트별로 수행한 결과를 그대로 반영하듯이 공간 범위 질의가 가장 빠른 처리 속도를 보여주었고, 다음으로 공간 KNN, 공간 조인 순으로 나왔다. 앞의 단일머신 테스트와 다르게 분산 환경에서의 데이터처리속도는 노드의 증가에 따라 그만큼 빠른 퍼포먼스를 보이듯이 분산 환경에서의 실험들은 노드 증가에 따른 처리성능을 보장한다는 것을 알 수 있었다.



[그림 23] 클러스터 실험 종합

결과적으로 공간 범위 질의는 다른 질의와는 다르게 셔플이 필요하지 않은 종속성만 가지고 인덱싱 된 SRDD를 캐싱하여 접근하기 때문에 제일 빠른 질의 처리가 가능 했을 것으로 볼 수 있다.

공간 KNN 질의의 경우 입력받는 값에 따라 연산 범위가 달라질 뿐 더러, 선택과 정렬 두 가지 단계를 수행하여 처리되기 때문에 공간 범위 다음으로 연산 수행 시간이 적게 들었을 것으로 볼 수 있다.

마지막으로 공간 조인 질의는 입력 받는 공간 조인 내에 포함 되는 모든 포인트와 폴리곤들을 반환하기 때문에 세 가지 질의 중에서 가잘 오랜 수행 시간이 걸렸을 거라 볼 수 있다.

제 5 장 결론 및 향후 계획

오늘날 우리는 디지털 세계에서 살고 있으며 다양한 영역에서 기하급수적인 데이터를 생산하고 있다. 이러한 현상을 '빅 데이터 시대'라고 한다. 특히 GPS를 내장하고 있는 모바일 장치의 보급이 보편화 되고, SNS가 발달되면서 일반 사용자들도 쉽게 위치 기반 서비스를 사용할 수 있게 되었다. 이를 통한 데이터의 증가는 기존의 공간 질의어로 처리하는 것이 어려워졌다.

따라서 본 논문에서는 대용량의 공간 데이터를 처리하기 위해 인-메모리 환경에서 분산 처리를 지원하는 프레임워크인 아파치 세도나를 제안하였으며, 이를 기반으로 한 시스템을 구축하였다. 또한 분산 환경에서 효율적으로 데이터를 처리하기 위하여 공간 정보를 기반으로 전체 데이터를 분할하여 파티션을 만들어주는 전역 그리드와 각 클러스터에 구축되는 지역 인덱스를 사용하였다. 전역 그리드에 사용하는 공간 분할 알고리즘에는 기존의 공간 빅 데이터 시스템에서 많이 쓰이는 공간 분할 알고리즘 중에 인-메모리 환경에 적합한 알고리즘을 제시하고, 세도나에 적용하였다.

결과적으로 연결된 클러스터의 수가 적을 때는 실행 시간이 급격히 증가 함을 볼 수 있지만 클러스터 수가 증가할수록 실행 시간의 감소율이 떨어진다. 그 이유로는 마스터 서버와 슬레이브 서버 각각에 대한 통신에 기본적인 시간이 소요되기 때문이다. 또한 슬레이브 서버의 수가 증가하면 연결된 슬레이브 서버 간의 통신이 증가하기 때문이다.

향후 연구로 각 서버간의 통신을 줄여 속도를 개선하는 추가적 연구와 그리

드를 이용하여 데이터를 분할할 때, 공간만 고려한 분할 방법이 아니라 단어를 고려한 분할 기법에 대한 연구가 필요하며, 공간 분석에 사용되는 다양한 프레임 워크를 대상으로 단일머신 및 분산클러스터에서 처리성능을 평가할 필요가 있다.

참 고 문 헌

[1] 김기련. "하둡 클러스터 시스템 구축과 SparkR을 이용한 시스템의 성능고찰." 국내석사학위논문 영남대학교 대학원, 2018. 경상북도

[2] http://www.nsdi.go.kr/lxportal/?menuno=2679 (국가공간정보포털)

[3] http://www.gisdeveloper.co.kr/?p=2332 (공간정보시스템 연구소)

[4] https://data.seoul.go.kr/dataList/OA-12252/S/1/datasetView.do (서울열린데이터광장)

[5] 김상호. "Apache Spark를 활용한 교통빅데이터 분석 및 처리성능 평가." 국내석사학위논문 충북대학교, 2015. 충청북도

[6] 양평우. "SKSpark : 공간 키워드 질의를 지원하는 분산 공간 컴퓨팅 프레임워크." 국내박사학위논문 群山大學校, 2017. 전라북도

[7] 이상용. "아파치 엘라스틱서치 기반 로그스태시를 이용한 보안로그분석 시스템." 국내석사학위논문 대전대학교 일반대학원, 2015. 대전

[8]https://www.researchgate.net/publication/328431673_Spatial_data_management_in_apache_spark_the_GeoSpark_perspective_and_beyond

[9]Huang, Z.; Chen, Y.; Wan, L.; Peng, X. GeoSpark SQL: An Effective Framework Enabling Spatial Queries on Spark. ISPRS Int. J. Geo-Inf. 2017, 6, 285

[10] SIGSPATIAL'15: 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems Seattle Washington November, 2015

감사의 글

인생을 살면서 제가 살아온 삶을 뒤돌아보니 자신에게 있어 학위 과정의 길은 학문의 길 보다는 어쩌면 더 큰 교훈을 얻었던 과정에 가깝지 않았나 싶습니다. 이제 비로소 학사 논문의 모든 과정을 마치고 짧지 않은 대학교 생활을 마무리하며 지난 시간들을 돌이켜보니 많은 아쉬움과 후회가 남습니다. 학업적 성취에 있어서의 아쉬움만이 아닌, 고마운 많은 분들께 감사의 마음을 제대로 전하지 못했기에 더욱 그러한 것 같습니다. 제가 이렇게 성장하기까지 오랜 시간이 걸렸지만 그 세월 속에서 직·간접적으로 힘이 되고 방향을 잡아주셨던 많은 분들께 감사의 말씀을 전하고자 합니다.

가장 먼저 이 논문의 연구계획에서부터 완성에 이르기까지 이론적 체계와 학문적 기틀을 바로잡아 주시고 자상한 가르침을 베풀어 주신 이상정 교수님께 깊은 감사를 드립니다. 또한, 논문 작성 과정에서 아낌없는 지도로 많은 가르침을 주신 홍인식 교수님께도 감사드리며 매 학기 마다 큰 열정으로 심도 있는 강의를 해주신 컴퓨터공학과 교수님들께도 감사드립니다.

연구 과정에서 마치 자신의 일처럼 도와주며 항상 옆에서 힘이 되어준 학우 고한설과 학교생활 중에 많은 추억과 보람을 함께 나누었던 컴퓨터 시스템 연구실 부원 김익환, 이인재, 김재진, 송희령, 윤여일, 이승하 학우들에게도 고마움을 전합니다. 또한, 고등학교부터 지금의 자리까지 함께 하며 큰 재미와 도움을 준 현지원 동기에게 고맙다는 말씀을 전하고 싶습니다. 앞으로 서로가 하고자 하는 분야에서 최고가 될 수 있도록 기원하겠습니다.

마지막으로 항상 사랑으로 키워주시고 부족한 자식을 믿어주신 부모님께 감사의 말씀을 드립니다. 언제나 제 편이 되어 힘을 주시고 바르게 생각하고 행동할 수 있도록 가르쳐주신 부모님께 누가 되지 않는 딸이 되기 위해 더욱 성장하도록 노력하겠습니다.