KIPS Tr. Software and Data Eng.
Vol.3, No.2 pp.73~80 pISSN: 2287-5905

http://dx.doi.org/10.3745/KTSDE.2014.3.2.73

Efficient Processing of an Aggregate Query Stream in MapReduce

Hyunjean Choi† · Ki Yong Lee††

ABSTRACT

MapReduce is a widely used programming model for analyzing and processing Big data. Aggregate queries are one of the most common types of queries used for analyzing Big data. In this paper, we propose an efficient method for processing an aggregate query stream, where many concurrent users continuously issue different aggregate queries on the same data. Instead of processing each aggregate query separately, the proposed method processes multiple aggregate queries together in a batch by a single, optimized MapReduce job. As a result, the number of queries processed per unit time increases significantly. Through various experiments, we show that the proposed method improves the performance significantly compared to a naive method.

Keywords : MapReduce, Aggregate Query, Query Stream

맵리듀스에서 집계 질의 스트림의 효율적인 처리 기법

최 현 진† · 이 기 용††

요 약

맵리듀스는 빅데이터 분석 및 처리에 널리 사용되는 프로그래밍 모델이다. 빅데이터 분석을 위해 흔히 사용되는 질의 중 하나는 집계 질의(aggregate query)이다. 본 논문에서는 여러 사용자가 동시에 여러 집계 질의를 계속해서 요청하는 경우, 맵리듀스를 사용하여 이들 질의를 효율적으로 처리하는 방법을 제안한다. 제안 방법은 각 집계 질의를 개별적으로 처리하지 않고, 여러 집계 질의를 묶어 하나의 최적화된 맵리듀스 잡(job)으로 만들어 일괄 처리한다. 그 결과로 제안 방법은 단순 방법에 비해 시간당 처리하는 질의 수를 크게 증가시킨다. 성능 평가를 통해, 제안 방법은 단순 방법에 비해 질의 처리 속도를 크게 향상시킴을 보인다.

키워드 : 맵리듀스, 집계 질의, 질의 스트림

1. 서 론

최근 들어 소셜 네트워크 서비스(Social Networking Service, SNS), 각종 웹 사이트(Web site), 환경/교통 모니터링, 센서 네트워크 등 다양한 분야에서 데이터가 급증하면서 소위 빅데이터(Big Data)[1]에 대한 관심이 커지고 있다. 빅데이터란 용량이 매우 크거나, 증가 속도가 매우 빠르거나, 형태가 매우 다양해서 현존하는 기술로는 빠르게 처리할 수 없는 데이터를 말한다[2].

빅데이터는 일반적으로 용량이 매우 크기 때문에 단일 컴퓨터로 처리하는 것이 불가능하다. 빅데이터 플랫폼은 대용량의 데이터를 다수의 컴퓨터에 분산하여 저장하고, 저장된 데이터를 다수의 컴퓨터로 병렬 처리할 수 있도록 해주는 플랫폼이다. 하둡(Hadoop)[3]은 현재 산업계 표준으로 사용되고 있는 대표적인 빅데이터 플랫폼이다. 하둡을 포함한 대부분의 빅데이터 플랫폼은 빅데이터를 병렬 처리하기 위해 맵리듀스(MapReduce)[4]라는 프로그래밍 모델을 제공한다.

맵리듀스를 사용하면 사용자는 자신이 수행하고자 하는 작업을 맵(Map)과 리듀스(Reduce)라는 두 함수로 표현한다. 그러면 시스템은 자동으로 다수의 컴퓨터를 사용하여 두 함수를 병렬적으로 수행한다. 더욱이 시스템은 작업 수행 중 어떤 컴퓨터에 고장이 발생하더라도, 해당 컴퓨터의 작업을 다른 컴퓨터에게 재할당하는 고장 감내(fault tolerance) 기능을 제공한다. 따라서 사용자는 복잡한 병렬 처리 메커니즘과 고장 감내 메커니즘을 전혀 모르고서도, 대용량의 데이터를 다수의 컴퓨터를 사용하여 병렬적으로 처리할 수 있

※ 이 논문은 2013년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No. 2012R1A1A1001269).
※ 이 논문은 제40회 한국정보처리학회 추계학술발표대회에서 '맵리듀스에서 집계 질의 스트림의 효율적인 처리 기법'의 제목으로 발표된 논문을 확장한 것임.
† 준 회 원 : 숙명여자대학교 컴퓨터과학부 학사과정
†† 정 회 원 : 숙명여자대학교 컴퓨터과학부 조교수
논문접수 : 2013년 12월 10일
수 정 일 : 1차 2014년 1월 14일
심사완료 : 2014년 1월 14일
* Corresponding Author : Ki Yong Lee(kiyonglee@sookmyung.ac.kr)

다. 이러한 사용의 용이성과 편리성으로 인해 맵리듀스는 빅데이터 처리의 표준 프로그래밍 모델로 자리 잡았다.

한편 빅데이터를 분석하기 위해 주로 사용되는 질의 중 하나는 집계(aggregate) 질의이다. 집계 질의는 데이터를 요약하는 집계치(예: 평균값, 최대값 등)를 구하는 질의이다. 집계 질의의 대표적인 예는 SQL의 집계 함수 *SUM*(), *COUNT*(), *AVG*(), *MAX*(), *MIN*() 등을 포함하는 질의로서, 각각 주어진 데이터에 대해 합, 개수, 평균값, 최대값, 최소값 등과 같은 집계치를 구한다.

최근 빅데이터의 개방이 활발해지고 인터넷을 통한 접근성이 향상되면서, 누구나 빅데이터에 접근하여 질의를 던지고 그 결과를 활용하도록 하는 기술에 대한 연구가 활발히 진행되고 있다. 예를 들어, 여러 사용자의 스마트폰 사용 내역(예: 웹 사이트 방문, 앱 사용 등)이 서버에 로그(log)로 저장되고 있다고 할 때, 각 사용자는 로그 데이터에 대한 집계 질의를 던져 자신과 동일한 위치, 동일한 시간대, 혹은 동일한 연령대에 있는 사용자들의 사용 패턴에 대한 정보를 얻을 수 있다. 이 경우는 로그라는 빅데이터에 대해 서로 다른 집계 질의가 계속해서 요청되는 예로 볼 수 있다.

본 논문에서는 여러 사용자가 동시에 서로 다른 집계 질의를 계속해서 요청하는 환경에서, 맵리듀스를 사용하여 이들 질의를 효율적으로 처리하는 방법을 제안한다. 앞서 언급한 바와 같이, 맵리듀스를 이용하면 질의 처리에 병렬 처리와 고장 감내가 자동으로 지원된다. 제안 방법은 각 집계 질의를 개별적으로 처리하는 대신, 여러 집계 질의를 묶어 하나의 최적화된 맵리듀스 잡(job)으로 만들어 일괄 처리한다. 그 결과로 제안 방법은 맵리듀스를 사용하는 단순 방법에 비해 시간당 처리하는 질의 수를 크게 증가시킨다. 또한 본 논문은 성능 평가를 위한 다양한 실험을 통해 제안 방법이 단순 방법에 비해 질의 처리 성능을 크게 향상시킴을 보인다. 본 논문은 [5]에서 발표한 방법을 확장한 것이다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 다루는 문제를 정의하고, 3장에서는 관련 연구를 간략히 살펴본다. 4장에서는 제안 방법을 자세히 설명하고, 5장에서는 성능 평가 결과를 설명한다. 마지막으로 6장에서는 결론을 맺는다.

2. 문제 정의

본 논문에서는 질의의 대상이 되는 데이터 D가 하나 이상의 튜플(tuple)로 이루어진, 다음과 같은 스키마(schema)를 가진 릴레이션(relation)이라고 가정한다.

$$D(A_1, A_2, \cdots, A_d, M)$$

여기에서 $A_1, A_2, \cdots, A_d$는 시간(timestamp), 사용자 ID, 위치 등과 같이 각 로그의 세부 정보를 담고 있는 애트리뷰트(attribute)이며, 본 논문에서는 모든 값이 숫자로 표현된다고 가정한다. M은 집계 대상이 되는 값을 저장하는 애트리뷰트이다. 여러 사용자는 각각 D에 대해 서로 다른 집계 질의를 던질 수 있다. 본 논문에서는 각 집계 질의 q가 다음 형태를 가진다고 가정한다.

$$q = (l_1 \le A_1 \le u_1,\ l_2 \le A_2 \le u_2,\ \cdots,\ l_d \le A_d \le u_d,\ AGG(M))$$

위 질의 q는 $A_1, A_2, \cdots, A_d$ 애트리뷰트의 값이 각각 $l_1 \le A_1 \le u_1$, $l_2 \le A_2 \le u_2$, …, $l_d \le A_d \le u_d$을 만족하는 튜플들에 대해 집계 함수 $AGG(M)$의 값을 구하는 질의이다. $c \le A_i \le c$는 $A_i = c$와 같음에 유의하라. 여기서 AGG는 *SUM*, *COUNT*, *AVG*, *MAX*, *MIN*과 같이 SQL에 정의된 집계 함수를 나타낸다. 본 논문에서는 설명의 편의를 위해 AGG는 *SUM*이라고 가정한다. 하지만 제안 방법은 *SUM* 외에 *COUNT*, *AVG*, *MAX*, *MIN*에도 동일하게 적용될 수 있다. $q = (A_1 = 10,\ 2 \le A_2 \le 5,\ 100 \le A_3 \le 200,\ SUM(M))$은 A_1, A_2, A_3 애트리뷰트의 값이 각각 $A_1 = 10$, $2 \le A_2 \le 5$, $100 \le A_3 \le 200$을 만족하는 튜플들에 대해 집계 함수 $SUM(M)$의 값을 요청하는 질의이다

본 논문에서는 q와 같은 형태의 집계 질의가 계속해서 요청되는 환경을 고려한다. 사용자들이 서버에 다양한 질의를 요청하면, 서버가 이들 중 q와 같은 형태의 질의를 모아 맵리듀스 프레임워크로 전송한다. 이렇게 계속해서 요청되는 일련의 집계 질의들을 집계 질의 스트림(stream)이라 부른다. 본 논문에서는 집계 질의 스트림을 맵리듀스를 사용하여 효율적으로 처리하는 방법에 대해 논의한다. 처리 방법의 성능은 해당 방법을 사용했을 때 시간당 처리되는 질의 수로 측정한다.

3. 배경 지식 및 관련 연구

3.1 맵리듀스

어떤 작업을 맵리듀스 환경에서 수행하기 위해서는 해당 작업을 맵과 리듀스 두 개의 함수로 표현해야 한다. 맵 함수는 하나의 키-값 쌍(key-value pair) (k, v)를 입력으로 받아 하나 이상의 키-값 쌍들 (k_1, v_1), (k_2, v_2), …을 출력으로 내보낸다. 입력으로 들어온 모든 키-값 쌍에 대해 맵 함수의 수행이 완료되면, 맵리듀스는 맵 함수가 출력한 모든 키-값 쌍들을 정렬하여 같은 키 값을 가진 쌍들 (k, v_1), (k, v_2), …을 모아 $(k, [v_1, v_2, \cdots])$ 형태로 만든 뒤, 이들을 리듀스 함수의 입력으로 전달한다. 리듀스 함수는 하나의 키-값 리스트 쌍 $(k, [v_1, v_2, \cdots])$을 입력으로 받아 하나 이상의 키 값 쌍들 (k_1, v_1'), (k_2, v_2'), …을 출력으로 내보내며, 리듀스 함수의 모든 출력을 모은 것이 맵리듀스의 최종 수행 결과가 된다.

맵리듀스에서 데이터는 분산 파일 시스템의 파일로 저장된다. 대용량 파일은 고정된 크기(주로 64 MB)로 분할되어 저장되며, 각 조각을 스플릿(split)이라 부른다. 각 스플릿은 고장 감내를 위해 하나 이상의 컴퓨터에 중복되어 저장된다.

맵리듀스에서 하나의 맵 함수와 하나의 리듀스 함수로 구

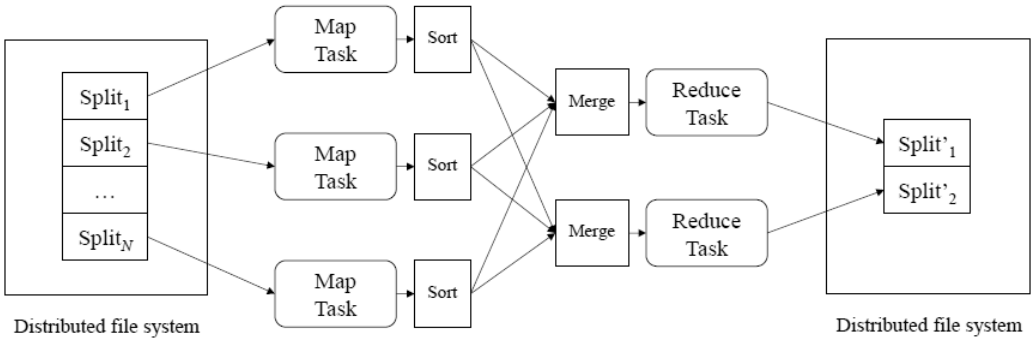

Fig. 1. Execution of a MapReduce job

성된 작업을 맵리듀스 잡(job)이라 부른다. Fig. 1은 맵리듀스 잡의 수행을 나타내는 그림이다. 어떤 맵리듀스 잡의 수행이 요청되면, 맵리듀스는 사용자가 지정한 수만큼의 맵 태스크(map task)와 리듀스 태스크(reduce task)를 생성한다. 각 맵 태스크는 맵 함수의 수행을 담당하며, 각 리듀스 태스크는 리듀스 함수의 수행을 담당한다. 각 맵 태스크는 입력 데이터를 저장하고 있는 파일의 한 스플릿을 할당받아, 해당 스플릿에 저장된 데이터를 사용자가 지정한 방식에 따라 키-값 쌍들로 변환하고, 각 키-값 쌍에 대해 사용자가 지정한 맵 함수를 호출한다. 맵 함수의 수행 결과로 생성된 키-값 쌍들은 키 값에 따라 정렬되어 디스크에 저장된다. 모든 맵 태스크의 수행이 완료되면, 각 리듀스 태스크는 각 맵 태스크에 접근하여 맵 태스크가 결과로 출력한 키-값 쌍들 중 자신이 할당받은 키 값을 가진 키-값 쌍들을 가져온다. 이 때, 어떤 키 값이 어떤 리듀스 태스크에게 할당되느냐는 사용자가 지정한 규칙에 따르며, 어떤 경우든 동일한 키 값을 가진 키-값 쌍들은 반드시 동일한 리듀스 태스크로 전송된다. 맵 태스크의 수행 결과로 생성된 모든 키-값 쌍들이 리듀스 태스크들로 전송되고 나면, 리듀스 함수의 수행이 시작된다. 각 리듀스 태스크는 자신이 가져온 키-값 쌍들로부터 동일한 키 값을 가지는 키-값 쌍들 (k, v_1), (k, v_2), …을 모아 $(k, [v_1, v_2, \cdots])$ 형태의 키-값 리스트 쌍을 생성하고, 각 키-값 리스트 쌍에 대해 사용자가 지정한 리듀스 함수를 호출한다. 리듀스 함수는 하나의 키-값 리스트 쌍을 입력으로 받아 하나 이상의 키-값 쌍들을 출력으로 내보낸다. 리듀스 함수의 수행 결과로 생성된 키-값 쌍들은 파일에 저장되며, 이것이 맵리듀스 잡의 최종 결과물이 된다.

3.2 맵리듀스를 사용한 질의 처리

3.1절에서 설명한 맵리듀스를 사용하여 다양한 형태의 질의를 효율적으로 처리하기 위한 연구가 많이 진행되고 있다. [6][7]은 맵리듀스로 동등 조인(equi-join)과 세타 조인(theta join)을 처리하는 방법을 제안하였다. 데이터의 특성에 따라 all pair partitioning join algorithm, repartition join algorithm, broadcast join algorithm, semi-join algorithm 등 다양한 조인 알고리즘을 제안하였으며, 이들은 서로 다르게 정의된 맵과 리듀스 함수를 사용한다. [8]은 다중 조인(multi-way join)을 하나의 맵리듀스 잡으로 처리하는 기법을 제안하였다.

[9][10]은 맵리듀스 환경에서 SQL을 효율적으로 처리하는 방법을 제안하였다. 이들은 SQL에 포함된 Select, Project, Join, Group-By, Order-By과 같은 명령어를 맵과 리듀스 함수로 변환한다. [9]는 사용자가 SQL 질의를 던지면, 이를 자동으로 (하나 이상의) 맵리듀스 잡으로 변환함으로써 사용자가 직접 맵과 리듀스 함수를 작성하지 않고도 맵리듀스에서 복잡한 SQL 질의를 실행할 수 있도록 하였다. [10]는 사용자가 던진 SQL 질의를 전통적인 DBMS와 맵리듀스 기술을 혼합하여 처리하는 시스템을 제안하였다.

[11][12][13][14]는 맵리듀스를 사용하여 연속 질의(continuous query)를 처리하는 기법을 제안하였다. 연속 질의는 입력 데이터가 계속해서 변하는 질의로서, 데이터가 변경되면 그의 결과를 계속해서 갱신해야 한다. [11]는 맵 함수의 출력을 파이프라인을 통해 리듀스 함수의 입력으로 바로 전달함으로써, 입력이 사전에 모두 주어지지 않고 계속해서 들어오는 경우에도 그때까지 받은 입력에 대한 질의 처리 결과를 점진적으로 내보낼 수 있도록 하였다. [12]는 입력에서 달라진 부분을 감지하여, 달라진 부분에 대해서만 맵리듀스 잡을 수행하여 질의 결과를 갱신하는 기법을 제안하였다. [13]는 연속적으로 들어오는 로그 데이터에 대한 질의를 맵리듀스 아키텍처를 사용하여 처리하는 시스템을 제안하였다. [14]은 연속 질의를 처리하기 위해 맵과 리듀스 함수 대신 맵과 업데이트라는 함수를 사용하는 Muppet이라는 시스템을 제안하였다. 이 시스템은 각 노드의 메모리에 Slate라고 부르는 상태 정보를 유지하고, 이를 업데이트 함수가 계속하여 갱신하는 방식으로 연속 질의를 처리한다. 하지만 이들 연구는 모두 질의는 하나로 고정되고 그의 입력 데이터가 변하는 환경에 관한 것이며, 본 논문에서 고려하는 서로 다른 여러 질의가 연속적으로 들어오는 환경과는 다르다.

[15][16]은 맵리듀스 환경에서 다중 질의(multiple queries)를 효율적으로 처리하는 방법을 제안하였다. [15]은 여러 맵리듀스 잡이 동시에 요청되었을 때, 그들 간에 맵 함수의 수행 또는 맵 함수의 출력을 공유함으로써 여러 맵리듀스 잡을 동시에 효율적으로 처리할 수 있는 방법을 제안하였

다. [16]은 [15]의 기법을 더 일반화하여 맵 함수의 수행 및 출력을 좀 더 다양한 상황에서 공유할 수 있도록 하는 한편, 어떤 맵리듀스 잡의 중간 결과를 임시 저장하여 다른 잡의 수행에 활용할 수 있도록 하였다. 하지만 이들 연구 모두 본 논문에서 고려하는 서로 다른 조건을 가진 여러 집계 질의가 연속적으로 들어오는 환경은 다루고 있지 않다.

3.3 맵리듀스를 사용하지 않는 질의 처리

최근에는 맵리듀스를 사용하지 않고 직접 HDFS와 같은 분산 파일 시스템에 접근하여 SQL을 처리하는 시스템들이 개발되고 있다. Cloudera Impala[17]는 HDFS 또는 HBase에 저장된 데이터에 대해 맵리듀스를 사용하지 않고 직접 SQL을 처리하는 자체 질의 실행 엔진을 구현하였다. Apache Tajo[18]는 한국에서 개발한 대용량 데이터 웨어하우스 시스템으로서, 역시 맵리듀스를 사용하지 않고 SQL 질의를 분산 실행시키는 방식을 사용한다. 그 외에도 Apache Drill, Stringer 등 다양한 SQL-on-Hadoop 솔루션들이 개발되고 있다. 하지만 이들 연구는 모두 단일 SQL 질의 처리에 관한 것이며, 여러 질의가 연속적으로 들어오는 환경에 대해서는 다루고 있지 않다. 따라서 본 논문에서는 2장에서 정의한 집계 질의 스트림을 효율적으로 처리하는 방법을 제안한다.

4. 제안 방법

본 장에는 2장에서 정의한 집계 질의 스트림을 처리하는 간단한 방법을 먼저 알아보고, 그의 단점을 최소화한 제안 방법을 자세히 설명한다.

4.1 단순 방법

2장에서 정의한 하나의 집계 질의는 맵리듀스에서 맵 함수의 출력이 리듀스 함수의 입력으로 전달될 때 내부적으로 정렬이 수행된다는 점을 이용하여 매우 간단히 처리할 수 있다. 주어진 하나의 집계 질의를 $q = (l_1 \le A_1 \le u_1,\ l_2 \le A_2 \le u_2,\ \cdots,\ l_d \le A_d \le u_d,\ SUM(M))$이라 하고, 그의 식별자(identifier)를 $q.id$로 나타내자. 데이터 D의 한 튜플을 $t = (a_1, a_2, \cdots, a_d, m)$이라 하자. 여기서 $a_i (i = 1, 2, \cdots, d)$는 애트리뷰트 A_i의 값을 나타내며, m은 애트리뷰트 M의 값을 나타낸다. 맵 함수는 데이터 D의 각 튜플 $t = (a_1, a_2, \cdots, a_d, m)$을 입력으로 받아, 만약 $a_1, a_2, \ldots, a_d$가 각각 $l_1 \le a_1 \le u_1,\ l_2 \le a_2 \le u_2,\ \cdots,\ l_d \le a_d \le u_d$을 만족하면 키-값 쌍 $(q.id, m)$을 출력으로 내보낸다. 맵리듀스는 맵 함수가 출력한 키-값 쌍들을 키 값으로 정렬하여 이들을 $(q.id, [m_1, m_2, \cdots])$ 형태의 키-값 리스트 쌍들로 만든 뒤, 이들을 리듀스 함수의 입력으로 전달한다. 리듀스 함수는 키-값 리스트 쌍 $(q.id, [m_1, m_2, \cdots])$을 입력으로 받아, 키-값 쌍 $(q.id, m_1 + m_2 + \cdots)$을 생성하여 그를 최종 결과로 내보낸다.

2장에서 정의한 집계 질의가 계속해서 요청되는 경우(즉, 집계 질의 스트림의 경우) 각각을 개별적으로 하나의 맵리듀스 잡으로 만들어 처리하면 매우 큰 비용이 발생한다. 각 맵리듀스 잡은 시작 비용(맵 태스크 및 리듀스 태스크 설정 비용 등), 입력 데이터를 읽는 비용 등이 들기 때문에 각각의 질의마다 하나의 맵리듀스 잡을 수행하는 것은 매우 비효율적이다. 따라서 본 논문에서는 미리 설정된 수인 P개의 질의가 새로 도착하면, 이들 P개의 질의를 묶어 하나의 맵리듀스 잡으로 일괄 처리한다.

새로 도착한 P개 질의의 집합을 $Q = \{q_1, q_2, \cdots, q_P\}$라고 하자. 이들을 묶어 하나의 맵리듀스 잡으로 한꺼번에 처리한다. 맵 함수는 이전과 동일하게 데이터 D의 각 튜플 $t = (a_1, a_2, \cdots, a_d, m)$을 입력으로 받아, 만약 $a_1, a_2, \cdots, a_d$가 어떤 질의 $q \in Q$의 조건을 만족하면 키-값 쌍 $(q.id, m)$을 출력으로 내보낸다. 여기서 $a_1, a_2, \cdots, a_d$가 Q에 속한 둘 이상의 질의의 조건을 만족하면 각 질의 q에 대해 키-값 쌍 $(q.id, m)$을 출력으로 내보낸다. 맵리듀스는 이들을 정렬하여 각 $q.id$에 대해 키-값 리스트 쌍 $(q.id, [m_1, m_2, \cdots])$을 생성한 뒤 이를 리듀스 함수에게 전달한다. 리듀스 함수는 키-값 리스트 쌍 $(q.id, [m_1, m_2, \cdots])$을 입력으로 받아, $(q.id, m_1 + m_2 + \cdots)$을 최종 결과로 출력한다. 지금까지 설명한 간단한 방법을 NAIVE라고 부르자. Fig. 2는 NAIVE 방법을 나타내는 의사코드이다.

하지만 NAIVE 방법은 다음과 같은 문제점을 가진다. 맵 태스크는 데이터 D에 포함된 튜플들 중 Q에 포함된 질의의 조건을 만족하는 모든 튜플에 대해 하나 이상의 키-값 쌍을 출력으로 내보낸다. 특히 튜플이 하나 이상 질의의 조건을 만족하는 경우, 해당 튜플에 대해서는 해당 튜플이 만족하는 질의 개수만큼의 키-값 쌍을 출력으로 내보낸다. 이들 키-값 쌍들은 정렬 작업을 거쳐 중간결과로 디스크에 저장되는 한편, 네트워크를 통해 리듀스 태스크로 전송된다. 따

```
Map(key: null, value: t = (a1, a2, ..., ad, m) ∈ D)
   for each q = (l1 ≤ A1 ≤ u1, l2 ≤ A2 ≤ u2, …, ld ≤ Ad ≤ ud, SUM(M)) ∈ Q such that
         l1 ≤ a1 ≤ u1, l2 ≤ a2 ≤ u2, …, ld ≤ ad ≤ ud do
      emit (q.id, m)

Reduce(key: q.id, value_list: [m1, m2, ..., ])
      emit (q.id, m1 + m2 + ...)
```

Fig. 2. NAIVE method

라서 Q에 포함된 질의의 조건을 만족하는 튜플 수가 많으면 많을수록 맵 함수가 출력한 키-값 쌍들을 정렬하고, 디스크에 쓰고, 네트워크로 전송하는 데 드는 비용이 커지게 된다. 또한 매 P개의 질의가 새로 도착할 때마다 이러한 맵 리듀스 잡을 반복해서 수행해야 하므로, 시스템의 질의 처리 성능, 즉, 시간당 처리하는 질의 수가 저하된다. 따라서 본 논문은 이러한 문제점을 최소화하는 방법을 제안한다.

4.2 제안 방법

앞 절에서 설명한 NAIVE 방법의 문제점을 개선하기 위해서는, 맵 함수가 출력으로 내보내는 키-값 쌍들의 개수를 최소화해야 한다. 즉, 맵 함수가 출력으로 내보내는 키-값 쌍들의 개수가 줄어들수록 이를 정렬하고, 디스크에 저장하고, 네트워크를 통해 리듀스 태스크로 전송하는 데 드는 비용이 줄어든다. 따라서 제안하는 방법은 맵 태스크가 Q에 포함된 질의의 조건을 만족하는 모든 튜플에 대해 하나 이상의 키-값 쌍을 출력하는 대신, 최소한의 키-값 쌍들만 출력으로 내보낸다. 따라 이들을 정렬하고, 디스크에 저장하고, 네트워크로 전송하는 데 드는 비용이 최소화된다. 다음은 제안하는 방법을 자세히 설명한다.

제안 방법은 P개의 질의 $Q = \{q_1, q_2, \cdots, q_P\}$가 새로 도착하면, 각 질의 $q_i(i = 1, 2, \cdots, P)$를 d차원 공간의 한 객체로 간주한다. 즉, 질의 $q = (l_1 \le A_1 \le u_1, l_2 \le A_2 \le u_2, \cdots, l_d \le A_d \le u_d, SUM(M))$는 d차원 공간에서 j차원($j = 1, 2, \cdots, d$)의 범위가 $[l_j, u_j]$인 공간 객체(spatial object)를 나타낸다. 제안 방법에서 각 맵 태스크는 수행을 시작하기 전, P개의 공간 객체 $q_1, q_2, \cdots, q_P$에 대한 공간 색인(spatial index)을 메모리에 생성한다. 이것은 나중에 데이터 D의 각 튜플 t에 대해, Q에 포함된 질의 중 t와 관련된 질의를 빠르게 찾는데 사용된다. 즉, D의 튜플 $t = (a_1, a_2, \cdots, a_d, m)$가 d차원 공간의 한 점 $(a_1, a_2, \cdots, a_d)$에 대응된다고 할 때, 공간 색인을 사용하면 해당 점을 포함하고 있는 공간 객체, 즉, $l_1 \le a_1 \le u_1, l_2 \le a_2 \le u_2, \cdots, l_d \le a_d \le u_d$을 만족하는 질의 $q = (l_1 \le A_1 \le u_1, l_2 \le A_2 \le u_2, \cdots, l_d \le A_d \le u_d, SUM(M))$들을 빠르게 찾을 수 있다. 이렇게 질의에 대해 색인을 생성해 놓고, 각 튜플에 대해 그와 관련된 질의들을 색인 탐색으로 찾는 기법을 질의 색인(query index)[19][20]이라 한다. 주어진 공간 객체에 대한 색인으로는 R-tree, R*-tree, quad-tree, grid index 등 다양한 공간 색인을 사용할 수 있다. 본 논문에서는 질의 색인을 위한 용도로 gird index를 사용한다. Grid index는 구조가 간단하여 매우 빠르게 구축할 수 있는 한편, 트리 구조 색인에 비해 트리가 깊어져 발생하는 탐색 속도 저하가 적다는 장점이 있다[19][20]. Fig. 3은 질의에 대해 생성된 grid index의 예를 보여준다. Grid index로 색인된 각 질의 객체는 그의 $SUM(M)$ 값을 담을 수 있는 저장 공간을 가지고 있으며, 각각 0으로 초기화된다.

각 맵 태스크가 맵 함수를 수행하기 전 $Q = \{q_1, q_2, \cdots, q_P\}$에 대해 구축한 질의 색인을 G라고 하자. 맵 함수는 데이터 D의 튜플 $t = (a_1, a_2, \cdots, a_d, m)$을 입력으로 받으면, G를 탐색하여 d차원 공간의 점 $(a_1, a_2, \cdots, a_d)$을 포함하고 있는 질의 객체들을 찾는다. 만약 그런 질의 객체 q를 찾으면 q의 $SUM(M)$ 값을 m만큼 증가시키고, 그런 객체가 없으면 아무것도 하지 않는다. 따라서 각 질의 객체는 현재까지 들어온 튜플에 대한 $SUM(M)$ 값을 유지하고 있게 된다. 여기서 맵 함수는 아무런 키-값 쌍을 출력하지 않음에 유의하자. 맵 태스크는 모든 입력 튜플에 대한 맵 함수의 수행이 완료되면, 메모리에 유지되고 있는 각 질의 객체 q에 대해 키-값 쌍 $(q.id, SUM(M))$을 생성하여 출력으로 내보낸다. 따라서 맵 태스크는 D의 튜플 중 Q에 포함된 질의의 조건을 만족하는 모든 튜플에 대해 키-값 쌍을 출력으로 내보내는 대신, Q에 포함되어 있는 각 질의 당 하나씩, P개의 키-값 쌍만을 출력으로 내보내게 된다. 맵리듀스는 맵 태스크가 출력한 키-값 쌍들을 정렬하여 각 $q.id$에 대해 키-값 리스트 쌍 $(q.id, [SUM(M)_1, SUM(M)_2, \cdots])$들을 생성한 뒤 이를 리듀스 함수에게 전달한다. 리듀스 함수는 키-값 리스트 쌍 $(q.id, [SUM(M)_1, SUM(M)_2, \cdots])$을 입력으로 받아 $(q.id, SUM(M)_1 + SUM(M)_2 + \cdots)$을 최종 결과로 출력한다. Fig. 4는 제안 방법을 나타내는 의사코드이다.

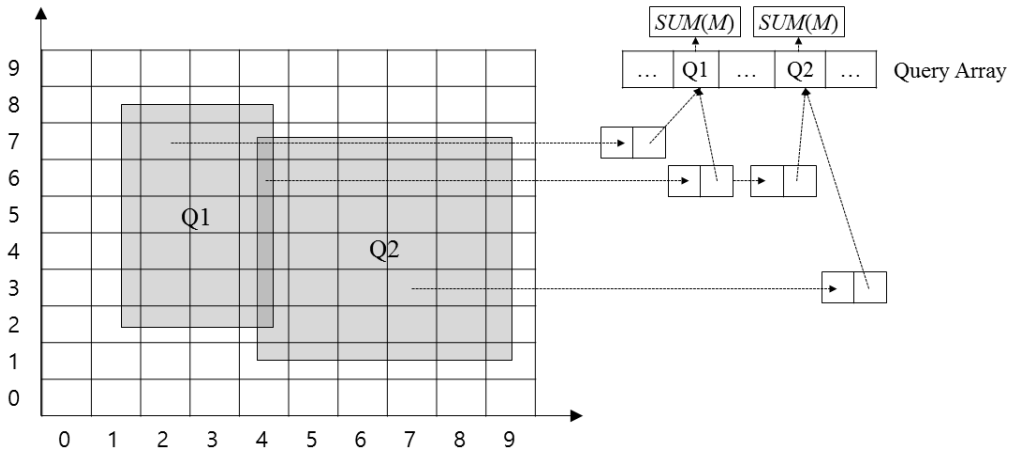

Fig. 3. Grid index for query indexing

```
Init()
   build a spatial index G on Q = {q1, q2, ..., qP}

Map(key: null, value: t = (a1, a2, ..., ad, m) ∈ D)
   retrieve the set Q' of all queries that contain the point (a1, a2, ..., ad) using G
   for each q ∈ Q' do
      increase its value of SUM(M) by m

Close()
   for each q ∈ Q do
      emit (q.id, SUM(M))

Reduce(key: q.id, value_list: [SUM(M)1, SUM(M)2, ..., ])
      emit (q.id, SUM(M)1 + SUM(M)2 + ...)
```

Fig. 4. The proposed method

제안 방법은 각 맵 태스크가 정확히 P개의 키-값 쌍만을 출력으로 내보내므로, 단순 방법에 비해 훨씬 적은 수의 키-값 쌍만을 출력으로 내보낸다. 이로 인해 맵 태스크의 결과를 정렬하는 비용, 디스크에 저장하는 비용, 리듀스 태스크로 전송하는 데 드는 비용이 크게 감소한다. 맵 함수의 수행도 데이터 D에 포함된 각 튜플에 대해 Q에 포함된 질의들을 각각 확인하여 그와 관련된 질의를 찾는 대신, grid index에 기반한 질의 색인 기법을 사용함으로써 관련된 질의를 찾는 비용이 크게 감소된다. 이에 따라 P개의 질의가 도착할 때마다 수행되는 맵리듀스 잡의 비용이 크게 줄어들어 시간당 처리되는 질의의 수가 향상된다. 또한 제안 방법에서 부가적으로 사용하는 grid index의 메모리 사용량은 색인 대상이 되는 질의 객체의 수가 100개일 때 수백 KB에 불과하므로 큰 비용을 요구하지 않는다[19].

마지막으로, 제안 방법에서는 P개의 질의가 새로 도착해야만 이들을 하나의 맵리듀스 잡으로 일괄 처리하기 때문에 최악의 경우에는 일찍 도착한 질의가 독립적으로 수행되는 경우보다 더 느리게 처리될 수 있다. 이를 위해 새로 도착한 질의의 개수가 P개가 되지 않더라도, 이들 중 최초로 도착한 질의가 도착한 후 사전에 정의된 시간 T가 경과되면 지금까지 도착한 질의들을 묶어 하나의 맵리듀스 잡으로 일괄 처리하는 방법을 사용할 수 있다. 이때 T는 지금까지 도착한 질의들의 평균 도착률 등을 사용하여 정할 수 있다.

5. 성능 평가

본 장에서는 제안 방법의 성능을 NAIVE 방법과 비교 평가한 결과를 제시한다. 성능 척도로는 동일한 개수의 질의를 처리하는 데 걸린 총 시간을 사용하였다. 시간당 처리되는 질의 수는 총 처리 시간을 질의 수로 나눔으로써 쉽게 구할 수 있다. 실험에서는 요청되는 집계 질의의 수를 1,000개에서 최대 5,000개로 늘려가며 모든 집계 질의 처리에 걸린 총 시간을 측정하였다. NAIVE 방법과 제안 방법 모두 100개의 질의가 도착할 때마다 하나의 맵리듀스 잡을 수행하였다.

실험에서 데이터 D는 가상 데이터를 생성하였으며, D의 각 튜플은 세부 정보를 나타내는 두 개의 애트리뷰트 A_1, A_2, 집계 대상이 되는 값을 저장하는 애트리뷰트 M, 임의의 값으로 채워지는 더미(dummy) 애트리뷰트로 이루어져있다. A_1과 A_2 값은 [0, 10,000] 범위에서 균등분포를 가지도록 임의로 생성하였으며, M 값은 [0, 100] 범위에서 임의로 생성하였다. 더미 애트리뷰트의 값은 임의의 문자열로 하였다. D의 각 튜플의 크기는 1 KB로 하였으며, D에 포함된 튜플 수는 200,000개에서 1,000,000개까지 변화시켰다. 이는 각각 200 MB와 1 GB 크기의 데이터에 해당한다. 각 집계 질의는 q = ($l_1 \le A_1 \le u_1$, $l_2 \le A_2 \le u_2$, $SUM(M)$)의 형태를 가지며 l_1, u_1, l_2, u_2의 값은 [0, 10,000] 범위에서 $l_1 \le u_1$, $l_2 \le u_2$ 관계가 만족되도록 임의로 선택하였다. 실험에서는 Amazon EC2 서비스[21]를 통해 6대의 컴퓨터로 구성한 클러스터를 사용하였다. 맵 태스크와 리듀스 태스크의 수는 각각 10개와 5개로 하였다.

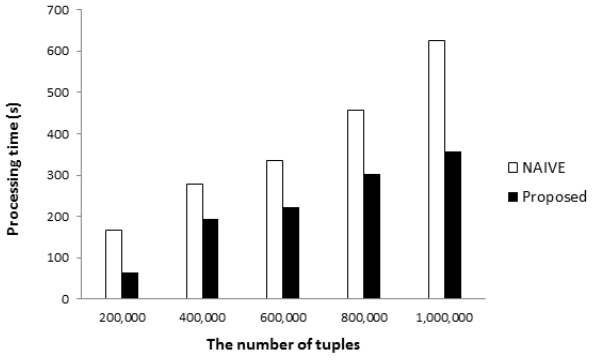

Fig. 5. Performance evaluation with varying number of tuples in D

Fig. 5는 데이터 D에 포함된 튜플 수를 변경시켜가며 두 방법의 성능을 비교한 결과이다. x축은 데이터 D에 포함된

총 튜플 수를 나타내며 200,000개에서 1,000,000개까지 증가시켰다. y축은 주어진 데이터에 대해 총 1,000개의 집계질의를 처리하는데 걸린 총 시간을 나타낸다. NAIVE 방법과 제안 방법 모두 100개의 질의가 새로 도착할 때마다 이를 묶어 하나의 맵리듀스 잡을 수행하였다. 4.2절에서 설명한 바와 같이 제안 방법은 단순 방법과 달리 맵태스크에서 데이터 D의 튜플 중 질의를 만족하는 모든 튜플들을 출력으로 내보내는 대신, 각 질의에 대해 하나씩의 키-쌍만을 출력으로 내보낸다. 이에 따라 맵 태스크에서 출력한 키-값 쌍들을 정렬하고, 디스크에 저장하고, 리듀스 태스크로 전달하는 비용을 크게 줄임으로써 전체 처리 성능을 크게 향상시켰음을 알 수 있다. 또한 제안 방법은 질의 색인 기법을 사용함으로써, 맵 태스크에서 데이터 D의 각 튜플에 대해 그와 관련된 질의를 찾는 시간을 줄여 질의 처리 시간을 더욱 감소시켰다

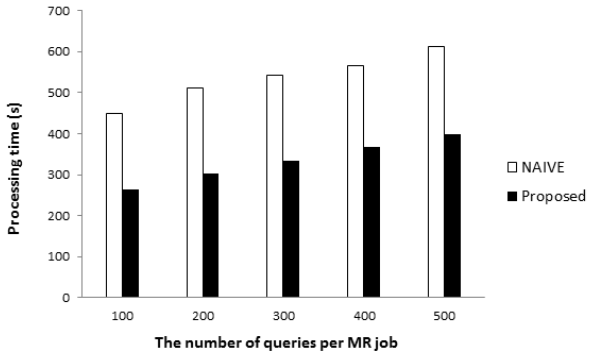

Fig. 6. Performance evaluation with varying number of queries per MR job

Fig. 6는 하나의 맵리듀스 잡이 동시에 처리하는 집계 질의의 수, 즉 P를 변화시켜가며 두 방법의 성능을 비교한 결과이다. 두 방법 모두 P를 100에서 500으로 증가시켜가며 성능을 측정하였으며, 각 P에 대해 맵리듀스 잡을 각각 10번씩 수행하였다. 즉, P = 100일 때는 100개의 집계질의가 새로 도착할 때마다 하나의 맵리듀스 잡을 수행하며, 총 100개 × 10번 = 1,000개의 집계질의를 처리한다. P = 500일 때는 500개의 집계질의가 새로 도착할 때마다 하나의 맵리듀스 잡을 수행하며, 총 500개 × 10번 = 5,000개의 집계질의를 처리한다. 하나의 맵리듀스 잡이 동시에 처리하는 질의 수가 증가할수록 NAIVE 방법은 맵 태스크가 중간결과로 출력하는 키-값 쌍의 양이 증가하기 때문에 (즉, 질의들을 만족하는 튜플 수가 증가하기 때문에) 처리 시간이 증가하게 된다. 동일한 상황에서 제안 방법 역시 맵 태스크가 출력해야 하는 키-값 쌍의 양이 증가하고 질의 탐색에 걸리는 시간이 증가하기 때문에 처리 시간이 증가한다. 하지만 Fig. 5에서 설명한 것과 동일한 이유로, 제안 방법은 NAIVE 방법에 비해 동일한 수의 질의를 처리하는데 훨씬 적은 시간이 드는 것을 확인할 수 있다.

Fig. 7은 데이터 D에 포함된 전체 튜플들 중, 하나의 맵리듀스 잡이 처리하는 P개 질의의 조건들을 만족하는 튜플들의 비율을 변화시켜가며 성능을 측정한 결과이다. 본 실험에서는 이 비율을 선택도(selectivity)라 부른다. 선택도가 1%인 경우, P개 질의를 처리하는 하나의 맵리듀스 잡이 수행될 때마다 데이터 D에 포함된 전체 튜플들 중 1%가 질의 결과에 참여한다. Fig. 7에서 볼 수 있듯이, NAIVE 방법은 선택도가 증가할수록 성능이 저하된다. 그 이유는 선택도가 증가할수록 맵 태스크가 출력으로 내보내는 키-값 쌍의 개수가 증가하기 때문이다. 이에 비해 제안 방법은 선택도에 의해 큰 영향을 받지 않음을 볼 수 있다. 이것은 제안 방법에서는 맵 태스크에서 선택도와 관계없이 언제나 P개 만큼의 키-값 쌍을 출력으로 내보내기 때문이다.

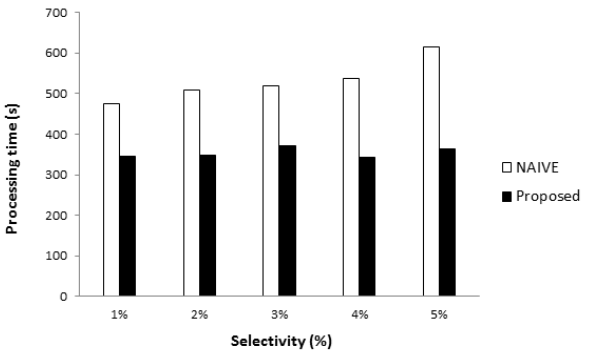

Fig. 7. Performance evaluation with varying selectivity

6. 결 론

본 논문에서는 빅데이터 처리에 널리 사용되는 맵리듀스 환경에서, 계속해서 유입되는 집계 질의 스트림을 효율적으로 처리하는 방법을 제안하였다. 제안 방법은 여러 집계 질의를 묶어 하나의 맵리듀스 잡으로 일괄 처리한다. 제안 방법은 단순 방법과 비교하여 맵 태스크가 출력으로 내보내는 키-값 쌍의 개수를 줄임으로써, 맵리듀스에서 내부적으로 수행되는 중간 결과의 정렬, 저장, 전송에 드는 비용을 크게 줄였다. 또한 질의 색인 기법을 이용하여 각 입력 튜플과 관련된 질의를 빠르게 찾도록 함으로써, 맵 함수의 수행시간을 크게 감소시켰다. 실험 결과 제안 방법은 단순 방법에 비해 동일한 수의 질의를 처리하는데 걸리는 시간을 최대 45%까지 감소시킴을 확인하였다. 추후 연구로는 더 다양한 형태의 질의 스트림을 맵리듀스뿐만 아니라 더 다양한 형태의 빅데이터 플랫폼에서 처리하는 방법을 연구할 예정이다.

참 고 문 헌

[1] http://en.wikipedia.org/wiki/Big_data

[2] Mark Beyer, “Gartner Says Solving ‘Big Data’ Challenge Involves More Than Just Managing Volumes of Data,” Gartner, 2011.

[3] http://hadoop.apache.org/
[4] Jeffrey Dean, Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters," In Proceedings of OSDI '04, pp.137-150, 2004.
[5] Hyunjean Choi, Ki Yong Lee, "Efficient Processing of an Aggregate Query Stream in MapReduce," Korea Information Processing Society Fall Conference, Nov., 2013.
[6] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian, "A comparison of join algorithms for log processing in MaPreduce," In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp.975-986, 2010.
[7] Alper Okcan, Mirek Ridedewald, "Processing Theta-Joins using MapReduce," In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pp.949-960, 2011.
[8] Foto N. Afrati, Jeffrey D. Ullman, "Optimizing Multiway Joins in a Map-Reduce Environment," IEEE Transactions on Knowledge and Data Engineering, Vol.23, No.9, pp.1282-1298, 2011.
[9] Hive [Internet], http://hive.apache.orge/
[10] HadoopDB [Internet], http://db.cs.yale.edu/hadoopdb/hadoopdb.html.
[11] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, Russell Sears, "MapReduce Online," In Proceedings of NSDI, Vol.10, No.4, p.20, 2010.
[12] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, Rafael Pasquini, "Incoop: MapReduce for Incremental Computations," In Proceedings of SOCC'11, 2011.
[13] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, Kenneth Yocum, "In-situ MapReduce for log processing," In Proceedings of USENIXATC'11, 2011.
[14] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, AnHai Doan, "Muppet: MapReduce-style processing of fast data," In Proceedings of the VLDB Endowment, Vol.5, Issue 12, pp.1814-1825, 2012.
[15] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, Nick Koudas, "MRShare: sharing across multiple queries in MapReduce," In Proceedings of the VLDB Endowment, Vol.3, Issue 1-2, pp.494-505, 2010.
[16] Guoping Wang, Chee-Yong Chan, "Multi-Query Optimization in MapReduce Framework," In Proceedings of the VLDB Endowment, Vol.7, No.3, pp.145-156, 2013.
[17] Cloudera Impala [Internet], http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.
[18] Tajo [Internet], http://tajo.incubator.apache.org/
[19] Dmitri V. Kalashnikov, Sunil Prabhakar, Susanne E. Hambrusch, "Main Memory Evaluation of Monitoring Queries Over Moving Objects," Distributed and Parallel Databases, Vol.15, No.2, pp.117-135, 2004.
[20] Xiaohui Yu , Ken Q. Pu , Nick Koudas, "Monitoring k-Nearest Neighbor Queries Over Moving Objects," In Proceedings of the 21st International Conference on Data Engineering, pp.631-642, 2005.
[21] Amazon Elastic Compute Cloud(Amazon EC2) [Internet], http://aws.amazon.com/ec2/

최 현 진

e-mail : gomsun09@naver.com
2009년~현 재 숙명여자대학교 컴퓨터과학부 학사과정
관심분야 : 데이터베이스, 빅데이터

이 기 용

e-mail : kiyonglee@sookmyung.ac.kr
1998년 KAIST 전산학과(학사)
2000년 KAIST 전산학과(석사)
2006년 KAIST 전산학전공(박사)
2006년~2008년 삼성전자 책임연구원
2008년~2010년 KAIST 전산학과 연구조교수
2010년~현 재 숙명여자대학교 컴퓨터과학부 조교수
관심분야 : 데이터베이스, 질의처리, 빅데이터, 데이터웨어하우스