KIPS Tr. Software and Data Eng.
Vol.3, No.11 pp.479~486 pISSN: 2287-5905

http://dx.doi.org/10.3745/KTSDE.2014.3.11.479

Efficient Computation of Data Cubes Using MapReduce

Ki Yong Lee[†] · Sojeong Park[††] · Eunju Park[†††] · Jinkyung Park[††] · Yeunjung Choi[††]

ABSTRACT

MapReduce is a programing model used for parallelly processing a large amount of data. To analyze a large amount data, the data cube is widely used, which is an operator that computes group-bys for all possible combinations of given dimension attributes. When the number of dimension attributes is n, the data cube computes 2^n group-bys. In this paper, we propose an efficient method for computing data cubes using MapReduce. The proposed method partitions 2^n group-bys into ${}_nC_{\lceil n/2 \rceil}$ batches, and computes those batches in stages using $\lceil n/2 \rceil$ MapReduce jobs. Compared to the existing methods, the proposed method significantly reduces the amount of intermediate data generated by mappers, so that the cost of sorting and transferring those intermediate data is reduced significantly. Consequently, the total processing time for computing a data cube is reduced. Through experiments, we show the efficiency of the proposed method over the existing methods.

Keywords : Data Cube, MapReduce, Big Data, Query Processing, OLAP

맵리듀스를 사용한 데이터 큐브의 효율적인 계산 기법

이 기 용[†] · 박 소 정[††] · 박 은 주[†††] · 박 진 경[††] · 최 연 정[††]

요 약

맵리듀스(MapReduce)는 대용량 데이터를 다수의 컴퓨터로 병렬 처리하는 데 사용되는 프로그래밍 모델이다. 데이터 큐브(Data Cube)는 대용량 데이터 분석에 널리 사용되는 연산자로서, 주어진 차원 애트리뷰트들의 모든 가능한 조합에 대한 group-by들을 계산한다. 차원 애트리뷰트의 개수가 n일 때, 데이터 큐브는 총 2^n개의 group-by를 계산한다. 본 논문은 맵리듀스를 사용하여 데이터 큐브를 효율적으로 계산하는 방법을 제안한다. 제안 방법은 2^n개의 group-by를 ${}_nC_{\lceil n/2 \rceil}$개의 그룹으로 분할하고, 이 그룹들을 $\lceil n/2 \rceil$개의 맵리듀스 잡(job)을 통해 단계적으로 계산한다. 제안 방법은 기존 방법에 비해 맵퍼(mapper)가 생성하는 중간결과의 크기를 크게 줄임으로써 중간결과의 전송 및 정렬에 드는 비용을 크게 줄인다. 그에 따라 데이터 큐브를 계산하는 총 수행시간이 크게 감소된다. 실험을 통해 제안 방법이 기존 방법에 비해 더 빠르게 데이터 큐브를 계산함을 보인다.

키워드 : 데이터 큐브, 맵리듀스, 빅데이터, 질의 처리, OLAP

1. 서 론

최근 들어 소셜 네트워크 서비스(SNS), 웹 사이트, 클릭스트림(clickstream) 데이터, 센서 네트워크, 바이오센서 데이터 등 다양한 분야의 데이터가 급증하면서 소위 빅데이터(Big Data)[1]에 대한 관심이 매우 커지고 있다. 빅데이터란 용량이 매우 크거나, 증가 속도가 매우 빠르거나, 형태가 매우 다양해서 현존 기술로는 효율적으로 처리하기 어려운 데이터를 말한다[2].

이러한 빅데이터를 쉽고 빠르게 처리하기 위해 맵리듀스(MapReduce)[3]라는 처리 기술이 제안되었다. 맵리듀스는 여러 컴퓨터에 분산되어 저장되어있는 데이터를 손쉽게 병렬 처리할 수 있도록 해주는 프로그래밍 모델이다. 맵리듀스를 사용하려면 사용자는 자신이 수행하고자 하는 작업을 맵(Map)과 리듀스(Reduce)라는 두 개의 함수로 표현해야 한다. 이렇게 한 쌍의 맵과 리듀스 함수로 표현된 작업을 맵리듀스 잡(job)이라 한다. 사용자가 맵리듀스 잡의 수행을 요청하면, 시스템은 사용자가 지정한 맵과 리듀스 함수를 자동적으로 여러 컴퓨터를 사용하여 병렬적으로 수행한다.

※ 본 연구는 숙명여자대학교 2012학년도 교내연구비 지원에 의해 수행되었음 (과제번호 1-1203-0242).
※ 이 논문은 2014년도 한국정보처리학회 춘계학술발표대회에서 '맵리듀스에서 데이터 큐브의 효율적인 계산 기법'의 제목으로 발표된 논문을 확장한 것임.
† 정 회 원 : 숙명여자대학교 컴퓨터과학부 부교수
†† 비 회 원 : 숙명여자대학교 컴퓨터과학부 학사과정
††† 비 회 원 : 숙명여자대학교 컴퓨터과학부 석사과정
Manuscript Received : July 29, 2014
First Revision : September 4, 2014
Accepted : September 4, 2014
* Corresponding Author : Ki Yong Lee(kiyonglee@sookmyung.ac.kr)

더욱이 작업 수행 도중 어떤 컴퓨터에 문제가 발생하더라도, 맵리듀스는 해당 컴퓨터가 수행하던 작업을 자동적으로 다른 컴퓨터로 이동시킴으로써 고장 감내(fault tolerance) 기능을 제공한다. 따라서 사용자는 복잡한 병렬 처리 메커니즘이나 고장 감내 기술을 전혀 모르고서도 대용량의 데이터를 여러 컴퓨터를 활용하여 손쉽게 처리할 수 있다. 이러한 사용의 편리성으로 인해 맵리듀스는 현재 대용량 데이터 처리의 표준 기술로 자리 잡았다.

한편 데이터 큐브(Data Cube)는 대용량의 데이터를 다차원으로 분석하는 데 널리 사용되는 연산자이다[4]. 주어진 차원 어트리뷰트(dimension attribute)들에 대해, 데이터 큐브는 그들의 모든 가능한 조합에 대한 group-by를 각각 계산한다. 예를 들어, 3개의 차원 어트리뷰트 *a*, *b*, *c*가 주어졌을 때, 다음 데이터 큐브 질의는 *a*, *b*, *c*의 모든 가능한 조합인 *abc*, *ab*, *ac*, *bc*, *a*, *b*, *c*, ∅의 각 조합에 대한 group-by를 각각 계산한다. (단, ∅는 그룹화 어트리뷰트가 없는 group-by를 나타낸다.)

```
SELECT a, b, c, SUM(m)
FROM F
CUBE BY a, b, c
```

여기서 CUBE BY는 데이터 큐브를 나타내는 연산자이다. 데이터 큐브를 구성하는 각 group-by를 큐보이드(cuboid)라 부른다. 따라서 위 질의로 정의된 데이터 큐브는 총 8개의 큐보이드로 구성된다. 일반적으로 차원 어트리뷰트의 개수가 n일 때, 데이터 큐브는 총 2^n개의 큐보이드를 계산한다. 따라서 데이터 큐브는 일반적으로 계산 비용이 매우 큰 연산자이다.

본 논문에서는 맵리듀스를 사용하여 데이터 큐브를 효율적으로 계산하는 방법을 제안한다. 맵리듀스를 사용하여 데이터 큐브를 효율적으로 계산하는 연구는 이미 진행된 바 있다[5, 6, 7, 8, 9]. 하지만 기존 방법은 2n개의 큐보이드 모두를 단 하나의 맵리듀스 잡으로 계산하는 과정에서 맵 함수가 출력으로 내보내는 중간결과의 크기가 매우 커진다는 단점이 있다. 이에 따라 맵리듀스가 이들을 디스크에 저장하고, 네트워크를 통해 리듀스 함수로 전송하고, 정렬하는 데 드는 비용이 커져 총 계산시간이 증가된다. 이에 비해 제안 방법은 2n개의 큐보이드를 ${}_nC_{\lceil n/2 \rceil}$개의 그룹으로 분할하고, 이 그룹들을 $\lceil n/2 \rceil$개의 맵리듀스 잡을 통해 단계적으로 계산한다. 이때 각 맵리듀스 잡에서 맵 함수가 출력으로 내보내는 중간결과의 크기를 최소화함으로써 각 맵리듀스 잡의 수행시간을 최소화한다. 따라서 제안 방법은 $\lceil n/2 \rceil$개의 맵리듀스 잡을 수행해야 하지만, 각 맵리듀스 잡의 수행시간이 감소되어 총 계산시간은 크게 줄어든다. 본 논문에서는 실제 계산시간을 측정한 실험을 통해, 제안 방법이 기존 방법에 비해 데이터 큐브를 계산하는 속도를 크게 향상시킴을 보인다.

본 논문의 구성은 다음과 같다. 2절에서는 데이터 큐브와 맵리듀스에 대한 배경 지식을 설명하고, 3절에서는 관련 연구를 살펴본다. 4절에서는 제안 방법을 설명하고, 5절에서는 실험 결과를 제시한다. 마지막으로 6절에서는 결론을 맺는다.

2. 배경 지식

2.1 데이터 큐브

데이터 큐브는 OLAP(Online Analytical Processing)에서 데이터를 다양한 차원으로 분석하는 데 널리 사용되는 연산자로서, 주어진 차원 어트리뷰트들의 모든 조합에 대한 group-by를 각각 계산한다. 이때 데이터 큐브를 구성하는 각 group-by를 큐보이드라 부른다. 예를 들어, 5개의 어트리뷰트를 가진 릴레이션 $F(a, b, c, d, m)$가 있을 때, 4개의 차원 어트리뷰트 *a*, *b*, *c*, *d*로 정의된 데이터 큐브는 총 2^4 = 16개의 큐보이드, 즉 *abcd*, *abc*, *abd*, *acd*, *bcd*, *ab*, *ac*, *ad*, *bc*, *bd*, *cd*, *a*, *b*, *c*, *d*, ∅로 구성된다.

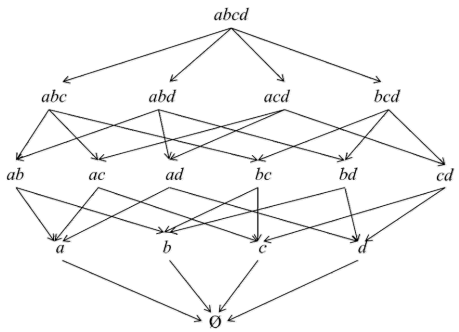

Fig. 1. Cube lattice

데이터 큐브는 흔히 큐브 격자(cube lattice)라 불리는 격자 다이어그램으로 표현된다[10]. Fig. 1은 앞서 예로 든 4개의 차원 어트리뷰트 *a*, *b*, *c*, *d*로 정의된 데이터 큐브를 표현하는 큐브 격자이다. 큐브 격자의 각 노드(node)는 큐보이드를 나타내며, 어떤 큐보이드 q_1에서 다른 큐보이드 q_2로 향하는 간선(edge)은 q_1으로부터 q_2를 계산할 수 있음을 나타낸다. 예를 들어, 큐보이드 *abc*에서 *ab*와 *ac*로 향하는 간선은 *abc*를 *a*, *b*와 *a*, *c* 어트리뷰트의 값으로 한 번 더 그룹화함으로써 *abc*로부터 *ab*와 *ac*를 구할 수 있음을 나타낸다.

2.2 맵리듀스

맵리듀스로 어떤 작업을 수행하려면 사용자는 해당 작업을 맵과 리듀스 두 개의 함수로 표현해야 한다. 이렇게 한 쌍의 맵과 리듀스 함수로 표현된 작업을 맵리듀스 잡이라 한다. 맵 함수는 입력으로 하나의 (key, value) 쌍 (k, v)를 받아 하나 이상의 (key, value) 쌍들 (k_1, v_1), (k_2, v_2), ...을 출력으로 내보내는 형태여야 하며, 리듀스 함수는 입력으로 하나의 (key, value list) 쌍 $(k, [v_1, v_2, ...])$을 받아 하나 이상의 (key, value) 쌍 (k'_1, v'_1), (k'_2, v'_2), ...을 출력으로 내보내는 형태여야 한다.

사용자가 맵리듀스 잡의 수행을 요청하면, 맵리듀스는 먼저 입력 데이터에 대해 사용자가 지정한 맵 함수를 수행한다. 이때 입력 데이터가 여러 대의 컴퓨터에 분산되어 저장되어있으면 맵 함수는 각각의 컴퓨터에서 병렬적으로 수행된다. 맵 함수의 수행을 담당하는 프로세스를 맵퍼(mapper)라고 한다. 입력 데이터에 대한 맵 함수의 수행이 모두 완료되면 맵리듀스는 맵 함수가 출력한 모든 (key, value) 쌍들을 모아 key값으로 정렬하고, 같은 key값을 가지는 (key, value) 쌍들을 모아 (key, value list) 쌍 (k, [v_1, v_2, ...])의 형태로 만든다. 그리고 이들을 리듀스 함수의 입력으로 전달한다.

이후 맵리듀스는 맵 함수가 출력한 (key, value) 쌍들로부터 생성된 모든 (key, value list) 쌍에 대해 사용자가 지정한 리듀스 함수를 호출한다. 이때 리듀스 함수도 사용자의 설정에 따라 여러 대의 컴퓨터에서 병렬적으로 수행될 수 있다. 최종적으로 리듀스 함수가 출력한 모든 (key, value) 쌍들의 집합이 맵리듀스 잡의 최종 수행결과가 된다. 리듀스 함수의 수행을 담당하는 프로세스를 리듀서(reducer)라고 한다.

일반적으로 맵리듀스 잡을 수행할 때 맵퍼가 출력으로 내보낸 (key, value) 쌍이 많으면 많을수록, 이들을 디스크에 저장하고, 네트워크를 통해 리듀서로 전달하고, 키값으로 정렬하는 데 드는 비용이 증가한다. 따라서 맵리듀스 잡의 수행시간을 줄이기 위해서는 맵퍼가 출력으로 내보내는 (key, value) 쌍의 수를 줄이는 것이 매우 중요하다.

3. 관련 연구

3.1 단순 방법

본 절에서는 맵리듀스를 사용하여 데이터 큐브를 계산하는 단순한 방법을 먼저 설명한다. 2.1절에서 예로 든 릴레이션 F(a, b, c, d, m)에 대해 다음 질의로 정의되는 데이터 큐브를 계산한다고 하자.

```
SELECT a, b, c, d, SUM(m)
FROM F
CUBE BY a, b, c, d
```

위 데이터 큐브는 4개의 차원 어트리뷰트 a, b, c, d로 정의되어있으며, 따라서 2^4 = 16개의 큐보이드로 구성된다. 맵 함수는 F의 한 튜플(tuple)을 입력으로 받아 각 큐보이드의 계산을 위해 하나씩, 총 16개의 (key, value) 쌍을 출력한다. 각 (key, value) 쌍의 key는 대응하는 큐보이드의 차원 어트리뷰트들의 값이며, value는 m 어트리뷰트의 값이다. 예를 들어, 입력으로 들어온 F의 한 튜플 (1, 2, 3, 4, 5)에 대해, 맵 함수는 ((1, 2, 3, 4), 5), ((1, 2, 3, *), 5), ((1, 2, *, 4), 5), ((1, *, 3, 4), 5), ... ((*, *, *, *), 5) 등 총 16개의 (key, value) 쌍을 출력으로 내보낸다. 각 (key, value) 쌍의 key는 각각 큐보이드 $abcd$, abc, abd, acd, ..., $\varnothing$의 차원 어트리뷰트들의 값을 나타내며, *는 해당 차원 어트리뷰트값이 사용되지 않음을 나타낸다. 리듀스 함수는 ((v_a, v_b, v_c, v_d, [v_{m1}, v_{m2}, ...]) 형태의 (key, value list) 쌍을 입력으로 받아 ((v_a, v_b, v_c, v_d, [v_{m1} + v_{m2} + ...]) 형태의 (key, value) 쌍을 출력으로 내보낸다. 리듀스 함수가 출력한 각 (key, value) 쌍은 데이터 큐브의 한 튜플을 나타내며, key와 value는 각각 해당 튜플의 a, b, c, d 어트리뷰트의 값과 SUM(m)의 값을 나타낸다.

이 방법은 단순하지만, 맵퍼가 F의 각 튜플에 대해 2^4 = 16개의 (key, value) 쌍을 출력해야 하므로 F의 튜플 개수를 $|F|$라 할 때, 맵퍼에 의해 총 16·$|F|$개의 (key, value) 쌍이 생성된다는 단점이 있다. 따라서 이들을 디스크에 저장하고, 네트워크를 통해 리듀서로 전달하고, key값으로 정렬하는 데 매우 큰 비용이 발생하게 된다.

3.2 배치 기반(batch-based) 방법

3.1절에서 설명한 단순 방법의 단점을 개선하기 위해, 배치(batch) 기반으로 데이터 큐브를 계산하는 연구가 진행되었다[5, 6].

[5]는 Fig. 2와 같이 2^n개의 큐보이드들을 여러 그룹으로 분할하고, 그룹별로 큐보이드들을 계산하는 방법을 제안하였다. 여기서 각 그룹을 배치(batch)라 부른다. Fig. 2의 b_1, b_2, ..., b_6는 각 배치의 ID를 나타낸다. 각 배치는 동일한 큐보이드를 조상(ancestor)으로 하는 큐보이드들이 모여 이루어진다. 예를 들어, Fig. 2의 배치 b_1에 속한 큐보이드 $abcd$, abc, ab, a, $\varnothing$들은 모두 $\varnothing$을 공통 조상으로 한다. 큐보이드들을 배치로 분할할 때는 어느 한 배치를 계산하는 리듀서에게 너무 많은 작업이 몰리지 않도록 하며, 다양한 분할 전략이 사용될 수 있다.

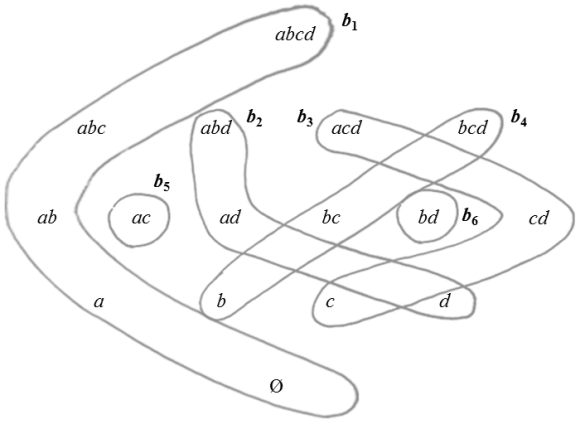

Fig. 2. Example of batches

2^n개의 큐보이드가 m개의 배치 b_1, b_2, ..., b_m로 분할되었다고 하자. 맵 함수는 F의 한 튜플을 입력으로 받아 각 배치의 계산을 위해 하나씩, 총 m개의 (key, value) 쌍을 출력한다. 각 (key, value) 쌍의 key는 대응하는 배치의 ID이며, value는 해당 튜플 자체다. 예를 들어, 2^n개의 큐보이드가

Fig. 2에서와 같이 6개의 배치 b_1, b_2, ..., b_6로 분할된 경우, 입력으로 들어온 F의 한 튜플 (1, 2, 3, 4, 5)에 대해, 맵 함수는 (b_1, (1, 2, 3, 4, 5)), (b_2, (1, 2, 3, 4, 5)), ..., (b_6, (1, 2, 3, 4, 5)) 등 총 6개의 (key, value) 쌍을 출력으로 내보낸다. 각각은 리듀스 함수가 배치 b_1, b_2, ..., b_6를 계산하는 데 사용된다. 리듀스 함수는 (b_i, [t_1, t_2, t_3, ...]) 형태의 (key, value list) 쌍을 입력으로 받아 튜플 t_1, t_2, t_3, ...로부터 배치 b_i에 속한 모든 큐보이드들을 계산하여 출력으로 내보낸다. t_1, t_2, t_3, ...로부터 여러 큐보이드들을 계산하기 위해 리듀스 함수는 BUC[11]와 같은 기존의 전통적인 데이터 큐브 알고리즘들을 사용할 수 있다. 만약 어떤 배치의 계산에 리듀서의 처리 용량을 넘는 튜플들이 몰리는 경우, 입력으로 들어온 튜플들을 값을 기준으로 분할하여 여러 리듀서에 분배한다.

이 방법은 맵퍼가 F의 각 튜플에 대해 배치 개수만큼의 (key, value) 쌍을 출력해야 하므로, 배치의 수를 m이라 했을 때 맵 함수에 의해 총 $m \cdot |F|$개의 (key, value) 쌍이 생성된다. 이것은 앞서 설명한 단순 방법에서 발생하는 $2^n \cdot |F|$개보다는 적은 수이긴 하지만, 여전히 배치의 수 m에 비례해서 맵리듀스 잡의 수행비용이 커지게 된다는 단점이 있다.

[6]는 [5]와 유사한 방법을 제안하였다. [6]는 2^n개의 모든 큐보이드들을 하나의 맵리듀스 잡으로 계산하기 위해서는 배치의 수가 최소 ${}_nC_{\lceil n/2 \rceil}$개 이상 되어야 한다는 것을 증명하였다. 이에 따라 [6]는 2^n개의 큐보이드들을 정확히 ${}_nC_{\lceil n/2 \rceil}$개의 배치로 분할하는 방법을 제안하였다. Fig. 2는 2^n개의 큐보이드를 정확히 ${}_nC_{\lceil n/2 \rceil}$개의 배치로 분할한 예에 해당한다. 이 방법에서도 맵 함수는 F의 한 튜플을 입력으로 받아 각 배치에 대해 하나씩, 총 ${}_nC_{\lceil n/2 \rceil}$개의 (key, value) 쌍을 출력한다. 따라서 맵퍼에 의해 총 ${}_nC_{\lceil n/2 \rceil} \cdot |F|$개의 (key, value) 쌍이 생성된다. 이 방법은 배치의 수를 최소화하여 맵 함수가 내보내는 (key, value) 쌍의 수를 줄이고, 그에 따라 맵리듀스 잡의 총 수행비용을 줄인다. 하지만 여전히 맵퍼에 의해 ${}_nC_{\lceil n/2 \rceil} \cdot |F|$개의 (key, value) 쌍들이 생성된다는 단점이 있다.

4. 제안 방법

본 절에서는 제안 방법을 설명한다. 제안 방법은 2^n개의 큐보이드를 ${}_nC_{\lceil n/2 \rceil}$개의 배치로 분할하고, 각 배치들을 $\lceil n/2 \rceil$개의 맵리듀스 잡을 통해 단계적으로 계산한다. 제안 방법은 $\lceil n/2 \rceil$개의 맵리듀스 잡을 사용하지만, 각 맵리듀스 잡의 수행비용을 최소화함으로써 총 수행시간을 크게 감소시킨다.

4.1 알고리즘

제안 방법은 먼저 2^n개의 큐보이드들을 ${}_nC_{\lceil n/2 \rceil}$개의 배치로 분할한다. *Level*($i$)를 차원 어트리뷰트가 i개인 큐보이드들의 집합이라 하자. 예를 들어, Fig. 1에서 *Level*(2) = {*ab*, *ac*, *ad*, *bc*, *bd*, *cd*}이다. 2^n개의 큐보이드들을 ${}_nC_{\lceil n/2 \rceil}$개의 배치로 분할하기 위해, 제안 방법은 *Level*(n)에서 *Level*(0)로 진행하며 *Level*(i)와 *Level*(i - 1)에 속한 큐보이드들 간의 최대 가중치 이분 매칭(maximum weight bipartite matching)을 구한다. (i = n, (n - 1), ..., 1)

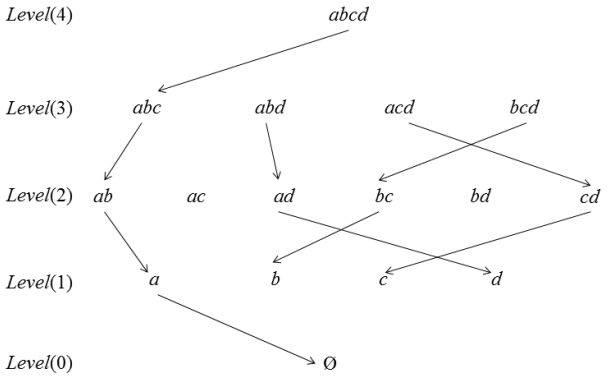

Fig. 3. Example of maximum weight bipartite matching

Fig. 3은 Fig. 1의 큐브 격자에서 *Level*(4)와 *Level*(3), *Level*(3)과 *Level*(2), *Level*(2)와 *Level*(1), *Level*(1)과 *Level*(0) 간의 최대 가중치 이분 매칭을 모두 구한 예를 보여준다. 이렇게 큐보이드들이 연결되고 나면, 연결된 큐보이드들이 모여 각각 하나의 배치를 형성한다. Fig. 2는 Fig. 3에서 최대 가중치 이분 매칭의 결과로 형성된 배치의 예에 해당한다. 이렇게 배치를 형성하면 배치는 큐보이드 수가 가장 많은 *Level*($\lceil n/2 \rceil$)의 큐보이드 수만큼 형성되므로, 배치의 개수는 정확히 *Level*($\lceil n/2 \rceil$)의 큐보이드 개수인 ${}_nC_{\lceil n/2 \rceil}$개가 된다. 이후 다음에 설명할 바와 같이, 동일한 배치에 속한 큐보이드들은 동일한 리듀서에 의해 같이 계산된다. 따라서 앞서 설명한 최대 가중치 이분 매칭을 구할 때 큐브 격자의 각 간선 e = (q_1, q_2)의 가중치 $w(e)$는 $|q_2|$로 둔다. 여기서 $|q|$는 큐보이드 q의 튜플 수를 나타낸다. 이것은 q_2와 q_1이 매칭되면 q_2는 q_1과 같이 계산되어 별도로 계산할 필요가 없어지므로, 가능한 크기가 큰 큐보이드를 기존의 배치에 포함시켜 별도로 계산할 필요를 없애기 위해서이다.

지금까지 설명한 방법에 따라 ${}_nC_{\lceil n/2 \rceil}$의 배치를 생성하고 나면, 제안 방법은 이들 배치를 $\lceil n/2 \rceil$개의 맵리듀스 잡을 통해 단계적으로 계산한다. 입력 데이터를 나타내는 릴레이션을 F라고 하자. 어떤 배치에 대해 그의 큐보이드들 중 차원 어트리뷰트의 수가 가장 많은 큐보이드를 헤드(head) 큐보이드라고 하자. 첫 번째 맵리듀스 잡은 F를 입력으로 하여, 헤드 큐보이드가 *Level*(n)에 속한 배치를 계산한다. 두 번째 맵리듀스 잡은 *Level*(n)을 입력으로 하여, 헤드 큐보이드가 *Level*(n - 1)에 속한 배치들을 계산한다. 동일한 방법으로 진행하여 마지막 $\lceil n/2 \rceil$번째 맵리듀스 잡은 *Level*($\lceil n/2 \rceil$ + 1)을 입력으로 하여, 헤드 큐보이드가 *Level*($\lceil n/2 \rceil$)에 속한 배치들을 계산한다. 앞서 설명한 배치 분할 방법을 사용하면 헤드 큐보이드가 i > $\lceil n/2 \rceil$인 *Level*(i)에 속한 배치는 있을 수 없으므로, $\lceil n/2 \rceil$번째 맵리듀스 잡까지 수행하고 나면 모든 배치의 계산이 완료되는 것이 보장된다.

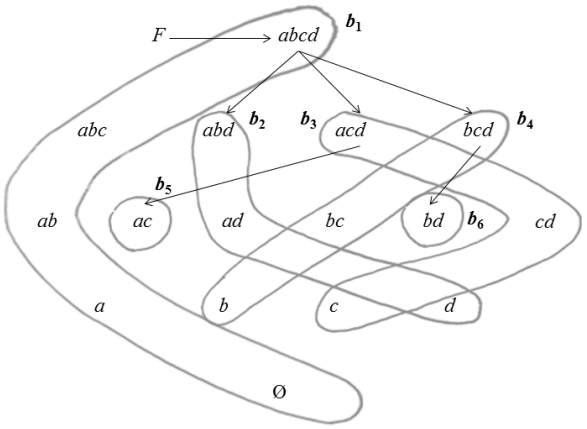

Fig. 4. Example of selecting a input cuboid for each batch

i번째 (i = 2, 3, ..., ⌈n/2⌉) 맵리듀스 잡을 좀 더 자세히 설명해보면, i번째 맵리듀스 잡은 $Level(n - i + 2)$를 입력으로 하여, 헤드 큐보이드가 $Level(n - i + 1)$에 속한 배치들을 계산한다. 헤드 큐보이드가 $Level(n - i + 1)$에 속한 배치들을 계산하기 위해서는 각 배치를 계산하는 데 입력으로 사용될 큐보이드를 $Level(n - i + 2)$에서 선택해야 한다. 헤드 큐보이드가 $Level(n - i + 1)$에 속한 어떤 배치를 b라 하자. b를 계산하는 데 입력으로 사용될 큐보이드는 b의 헤드 큐보이드와 큐브 격자에서 간선으로 연결된 $Level(n - i + 2)$의 큐보이드들 중 가장 크기가 작은 큐보이드를 선택한다. 이것은 입력 큐보이드의 크기가 작을수록 입력 큐보이드를 읽는 비용과 맵퍼가 출력으로 내보내는 (key, value) 쌍의 수가 줄어들어 해당 배치를 계산하는 비용이 최소화되기 때문이다. Fig. 4는 Fig. 2의 각 배치 b_2, b_3, ..., b_6를 계산하는 데 입력으로 사용될 큐보이드들을 선택한 예를 보여준다. (단, 배치 b_1은 F를 입력으로 사용한다.) Fig. 4는 b_1은 F로부터, b_2, b_3, b_4는 $abcd$로부터, b_5는 acd로부터, b_6는 bd로부터 계산된다는 것을 나타낸다.

이러한 방법으로 헤드 큐보이드가 $Level(n - i + 1)$에 속한 배치들을 계산하는 데 입력으로 사용될 $Level(n - i + 2)$의 큐보이드들이 각각 선택되었다고 하자. Fig. 5는 제안 방법의 맵과 리듀스 함수를 나타낸다. 맵 함수는 $Level(n - i + 2)$에서 선택된 큐보이드의 한 튜플 t를 입력으로 받아, 해당 큐보이드로 계산할 각 배치 b_i에 대해 하나씩의 (key, value) 쌍을 출력으로 내보낸다. 각 (key, value) 쌍의 key는 대응하는 배치의 ID이며, value는 해당 튜플 자체다. 예를 들어, Fig. 4에서 3번째 맵리듀스 잡은 acd를 입력으로 하여 배치 b_5를, bcd를 입력으로 하여 배치 b_6를 계산한다. 이를 위해 맵 함수는 acd의 튜플 t에 대해서는 (b_5, t)를, bcd의 튜플 t에 대해서는 (b_6, t)를 출력으로 내보낸다. 리듀스 함수는 (b_i, [t_1, t_2, t_3, ...]) 형태의 (key, value list) 쌍을 입력으로 받아 튜플 t_1, t_2, t_3, ...로부터 배치 b_i에 속한 모든 큐보이드들을 계산하여 출력으로 내보낸다.

```
Map(t)
1:  /* t is a tuple in the input data */
2:  Let B be the set of batches
3:  for each b_i in B
4:      emit(b_i.ID, t)

Reduce(key, value_list)
1:  Let B be the batch whose ID is key
2:  Let C = {c1, c2, ...} be the set of cuboids in B
3:  Let T = {t1, t2, ...} be the tuples in value_list
4:  result = computeCuboids({c1, c2, ...}, {t1, t2, ...})
5:  for each r in result
6:      emit(r)
```

Fig. 5. Map and Reduce functions of the proposed method

Fig. 4를 통해 제안 방법에서 ⌈n/2⌉개의 맵리듀스 잡을 수행하는 예를 설명한다. Fig. 4에서와 같이 각 배치를 계산할 입력 큐보이드가 결정되면, 첫 번째 맵리듀스 잡은 F를 입력으로 하여 헤드 큐보이드가 $Level(4)$에 속한 배치, 즉 b_1 = {$abcd$, abc, ab, a, ∅}를 계산한다. 두 번째 맵리듀스 잡은 $Level(4)$에 속한 $abcd$를 입력으로 하여 헤드 큐보이드가 $Level(3)$에 속한 배치, 즉 b_2 = {abd, ad, d}, b_3 = {acd, cd, c}, b_4 = {bcd, bc, b}를 각각 계산한다. 세 번째 맵리듀스 잡은 $Level(3)$에 속한 acd와 bcd를 입력으로 하여 헤드 큐보이드가 $Level(2)$에 속한 배치, 즉 b_5 = {ac}와 b_6 = {bd}를 각각 계산한다. 따라서 3개의 맵리듀스 잡으로 2^4 = 16개의 모든 큐보이드를 계산하게 된다.

제안 방법은 3개의 맵리듀스 잡을 사용하지만, $abcd$, abd, acd, bcd 등 F에 비해 크기가 매우 작은 큐보이드들을 사용하여 다른 큐보이드들을 계산하므로 맵퍼가 생성하는 (key, value) 쌍의 개수를 크게 줄인다. 따라서 4.2절에서 분석할 바와 같이 각 맵리듀스 잡의 수행비용이 줄어들어 총 수행비용이 크게 감소된다.

마지막으로 제안 방법이 데이터 큐브를 올바르게 계산함을 설명한다. 먼저 제안 방법은 2^n개의 큐보이드들을 여러 배치로 분할한다. 각 큐보이드는 이분 매칭 알고리즘에 따라 $Level(⌈n/2⌉)$에 속한 큐보이드 하나와 반드시 연결된다. 따라서 각 큐보이드는 $Level(⌈n/2⌉)$에 속한 ${}_nC_{⌈n/2⌉}$개의 큐보이드들 중 반드시 하나와 같은 배치에 속하게 된다. 따라서 각 큐보이드는 ${}_nC_{⌈n/2⌉}$개의 배치 중 하나에 반드시 속한다는 것이 증명된다. 다음으로 제안 방법은 ${}_nC_{⌈n/2⌉}$개의 배치들을 ⌈n/2⌉개의 맵리듀스 잡을 통해 계산한다. i번째 (i = 2, 3, ..., ⌈n/2⌉) 맵리듀스 잡은 ${}_nC_{⌈n/2⌉}$개의 배치 중 헤드 큐보이드가 $Level(n - i + 1)$에 속한 배치들을 계산하며, 앞서 설명한대로 헤드 큐보이드가 i > ⌈n/2⌉인 $Level(i)$에 속한 배치는 있을 수 없으므로, ⌈n/2⌉개의 맵리듀스 잡을 수행하면 모든 배치의 계산이 완료됨을 알 수 있다. 따라서 ${}_nC_{⌈n/2⌉}$의 배치로 분할된 2^n개의 모든 큐보이드들이 올바르게 계산됨이 증명된다.

4.2 비용 분석

본 절에서는 단순 방법(NAIVE), 배치 기반 방법(BATCH), 제안 방법(PROPOSED)의 수행비용을 비교한다. 이를 위해 Fig. 1의 데이터 큐브를 각 방법으로 계산하는 경우를 예로 들어 비교한다.

NAIVE는 맵퍼가 F의 각 튜플에 대해 2^4 = 16개씩, 총 $16 \cdot |F|$개의 (key, value) 쌍을 출력한다. 이들은 맵리듀스 엔진에 의해 정렬되어 리듀서로 전송된다. [12]에서 제시한 맵리듀스 잡의 비용 모델에 따르면, 이때의 비용 C_{NAIVE}는 대략 다음과 같이 예측된다.

$$C_{NAIVE} = C_1 \cdot |F| + C_2 \cdot (16 \cdot |F| \cdot \log(16 \cdot |F|)) + C_3 \cdot (16 \cdot |F|)$$

여기서 C_1는 디스크에서 튜플을 읽는 비용을 나타내는 비례상수이고, C_2는 (key, value) 쌍들을 정렬하는 비용을 나타내는 비례상수이며, C_3는 (key, value) 쌍들을 디스크에 쓰고 리듀서로 전송하는 데 드는 비용을 나타내는 비례상수이다. 첫 번째 항은 맵퍼가 입력 데이터를 읽는 비용을 나타내고, 두 번째 항은 맵퍼가 출력한 (key, value) 쌍들을 정렬하는 비용을 나타내며, 세 번째 항은 맵퍼가 출력한 (key, value) 쌍들을 디스크에 쓰고, 리듀서로 전송하는 데 드는 비용을 나타낸다. 최종 결과를 디스크에 쓰는 비용은 모든 방법에 공통되므로 비교를 위한 비용 모델에는 포함시키지 않았다.

BATCH는 맵퍼가 F를 읽어 각 튜플에 대해 배치 개수만큼의 (key, value) 쌍을 출력한다. Fig. 2에서와 같이 배치의 개수가 최소 개수인 ${}_4C_{\lceil 4/2 \rceil}$ = 6개라고 하자. 이 경우 맵퍼에 의해 총 $6 \cdot |F|$개의 (key, value) 쌍이 출력된다. NAIVE와 동일한 비용 모델을 사용하면 이 경우 BATCH의 비용 C_{BATCH}는 대략 다음과 같이 예측된다.

$$C_{BATCH} = C_1 \cdot |F| + C_2 \cdot (6 \cdot |F| \cdot \log(6 \cdot |F|)) + C_3 \cdot (6 \cdot |F|)$$

마지막으로 PROPOSED는 3개의 맵리듀스 잡을 수행한다. 3개의 맵리듀스 잡이 Fig. 4와 같이 수행된다고 하자. 첫 번째 잡은 F를 읽어 배치 {*abcd*, *abc*, *ab*, *a*, ∅}을 계산하며, 그 비용은 대략 $C_{PROPOSED1} = C_1 \cdot |F| + C_2 \cdot (|F| \cdot \log|F|) + C_3 \cdot |F|$로 예측된다. 두 번째 잡은 *abcd*를 읽어 배치 {*abd*, *ad*, *d*}, {*acd*, *cd*, *c*}, {*bcd*, *bc*, *b*}를 각각 계산하며, 그 비용은 대략 $C_{PROPOSED2} = C_1 \cdot |abcd| + C_2 \cdot (3 \cdot |abcd| \cdot \log(3 \cdot |abcd|)) + C_3 \cdot (3 \cdot |abcd|)$로 예측된다. 마지막으로 세 번째 잡은 *acd*와 *bcd*를 읽어 *acd*로는 배치 {*ac*}를, *bcd*로는 배치 {*bd*}를 각각 계산하며, 그 비용은 대략 $C_{PROPOSED3} = C_1 \cdot (|acd| + |bcd|) + C_2 \cdot ((|acd| + |bcd|) \cdot \log(|acd| + |bcd|)) + C_3 \cdot (|acd| + |bcd|)$로 예측된다. 따라서 PROPOSED의 총비용 $C_{PROPOSED}$는 대략 다음과 같다.

$$\begin{aligned} C_{PROPOSED} &= C_{PROPOSED1} + C_{PROPOSED2} + C_{PROPOSED3} \\ &= C_1 \cdot (|F| + |abcd| + |acd| + |bcd|) \\ &+ C_2 \cdot (|F| \cdot \log|F| + 3 \cdot |abcd| \cdot \log(3 \cdot |abcd|) \\ &\quad + (|acd| + |bcd|) \cdot \log(|acd| + |bcd|)) \\ &+ C_3 \cdot (|F| + 3 \cdot |abcd| + |acd| + |bcd|) \end{aligned}$$

예를 들어, $|F| = 10^6$, $|abcd| = 10^4$, $|acd| = |bcd| = 10^2$이라고 가정하자. 이 경우 PROPOSED는 NAIVE, BATCH 방법에 비해 입력을 읽는 비용이 $10^6 \cdot C_1$에서 $(10^6 + 10^4 + 2 \cdot 10^2) \cdot C_1$로 약간 증가하지만, 맵퍼가 출력한 (key, value) 쌍들을 정렬하는 비용은 NAIVE의 $16 \cdot 10^6 \cdot \log(16 \cdot 10^6) \cdot C_2 \approx 10^8 \cdot C_2$나 BATCH의 $6 \cdot 10^6 \cdot \log(6 \cdot 10^6) \cdot C_2 \approx 4 \cdot 10^7 \cdot C_2$에 비해 훨씬 적은 $(10^6 \cdot \log \cdot 10^6 + 3 \cdot 10^4 \cdot \log(3 \cdot 10^4) + 2 \cdot 10^2 \cdot \log(2 \cdot 10^2))C_2 \approx 6 \cdot 10^6 \cdot C_2$가 된다. 또한 맵퍼가 출력한 (key, value) 쌍들을 디스크에 쓰고, 리듀서로 전송하는 비용도 NAIVE의 $16 \cdot 10^6 \cdot C_3$나 BATCH의 $6 \cdot 10^6 \cdot C_3$에 비해 훨씬 적은 $(10^6 + 3 \cdot 10^4 + 2 \cdot 10^2)C_3 \approx 10^6 \cdot C_3$이 된다. 따라서 제안 방법은 맵퍼가 발생시키는 중간결과를 최소화하여 총 수행비용을 크게 줄임을 알 수 있다.

5. 실험 결과

본 절에서는 제안 방법의 성능을 기존 방법과 비교한 실험 결과를 보인다. 성능 척도로는 동일한 데이터 큐브를 계산하는 데 걸리는 총 수행시간을 비교하였다. 실험은 Amazon EC2 서비스[13]를 사용하였으며, 총 10개의 노드로 구성된 클러스터를 사용하였다. 각 노드는 Intel Xeon 프로세서 2.5 GHz CPU와 1 GiB 메모리를 탑재하고 있다. 맵 태스크와 리듀스 태스크의 수는 각각 10개로 하였다.

실험 데이터로는 가상의 데이터를 생성하여 사용하였다. 데이터를 저장하고 있는 릴레이션 F는 6개의 어트리뷰트 *a*, *b*, *c*, *d*, *m*, *dummy*로 이루어져 있으며, *a*, *b*, *c*, *d*는 차원 어트리뷰트, *m*은 집계의 대상이 되는 값을 담고 있는 측정(measure) 어트리뷰트, *dummy*는 임의의 값을 담는 어트리뷰트이다. *a*, *b*, *c*, *d*, *m*의 크기는 각각 4bytes, dummy는 30bytes이며, 따라서 F의 한 튜플의 크기는 50bytes가 된다. 본 절에서는 F의 튜플 수를 10^5, 10^6, 10^7개로 증가시켜가며 실험을 수행하였으며, *a*, *b*, *c*, *d*, *m*의 값은 각각 [1, 100]의 범위에서 균등 분포를 임의로 생성하였다. 데이터 큐브로는 4개의 어트리뷰트 *a*, *b*, *c*, *d*를 차원 어트리뷰트로 하는 데이터 큐브를 생성하였으며, 이는 총 2^4 = 16개의 큐보이드로 구성된다. 실험은 2회 수행하여 평균값을 취하였다.

Fig. 6는 F의 튜플 수가 10^5개, 10^6개, 10^7개일 때, 즉 $|F|$

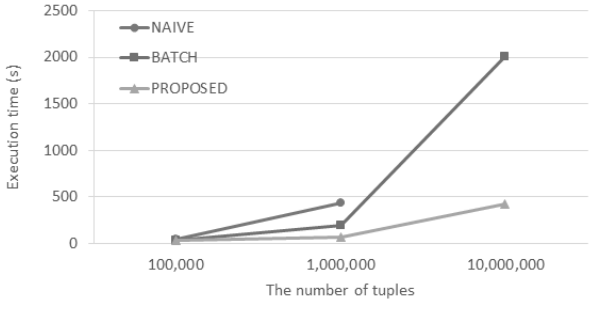

Fig. 6. Performance evaluation with varying number of tuples

= 10^5, 10^6, 10^7일 때, 3.1절에서 설명한 단순 방법(NAIVE), 3.2절에서 설명한 배치의 수를 최소화하는 [6]의 배치 기반 방법(BATCH), 4절에서 설명한 본 논문의 제안 방법(PROPOSED)으로 데이터 큐브를 계산하는 데 걸린 총 시간을 보인다. 3절에서 설명한 바와 같이, NAIVE는 맵퍼가 F의 각 튜플에 대해 2^4 = 16개씩 총 16·|F|개의 (key, value) 쌍을 중간결과로 출력한다. 따라서 이들을 디스크에 저장하고, 네트워크로 전송하고, 정렬하는 데 많은 비용이 발생한다. 한편 BATCH는 F의 각 튜플에 대해 ${}_4C_{\lceil 4/2 \rceil}$ = 6개씩 총 6·|F|개의 (key, value) 쌍을 중간결과로 출력하므로, NAIVE에 비해 맵리듀스 잡의 수행비용이 줄어들어 더 빠르게 데이터 큐브를 계산할 수 있다. 하지만 위 두 방법과 달리 PROPOSED는 3개의 맵리듀스 잡을 사용하지만 NAIVE와 BATCH에 비해 좋은 성능을 보인다.

다만 |F| = 10^5일 때 PROPOSED는 BATCH에 비해 크게 좋은 성능을 보이지 못하고 있다. 이것은 첫 번째 맵리듀스 잡에서 F로부터 계산된 abcd의 크기가 F에 비해 크기가 크게 줄어들지 않아서, 두 번째 맵리듀스 잡에서 *abcd*로부터 *abd*, *acd*, *bcd*를 계산할 때 F로부터 바로 계산하는 것에 비해 큰 이득을 얻지 못했기 때문이다. 이렇게 *abcd*의 크기가 F에 비해 크게 줄어들지 않은 이유는 *a*, *b*, *c*, *d* 어트리뷰트가 가질 수 있는 서로 다른 값이 각각 100개씩인데 F의 튜플 수가 10^5개에 불과하기 때문에 F에서 서로 다른 *a*, *b*, *c*, *d* 값을 가지는 튜플들이 그리 많지 않기 때문이다. 하지만 |F| = 10^6, 10^7일 때 볼 수 있듯이 F의 튜플 수가 많아지면 많아질수록 제안 방법의 성능이 크게 향상된다. |F| = 10^6일 때는 F에서 서로 다른 *a*, *b*, *c*, *d*값을 가지는 튜플들이 |F| = 10^5일 때에 비해 많이 생기기 때문에 F에 비해 *abcd*의 크기가 많이 작아지게 된다. 따라서 *abcd*에서 *abd*, *acd*, *bcd*를 계산하는 두 번째 잡의 비용이 F로부터 *abd*, *acd*, *bcd*를 직접 계산하는 것에 비해 크게 줄어들게 된다.

마지막으로 |F| = 10^7일 때 NAIVE는 맵리듀스 잡이 끝까지 수행되지 못하고 중간에 실패하였다. 이것은 NAIVE에서 맵퍼가 출력하는 중간결과의 크기가 너무 커져 리듀서가 감당할 수 있는 처리용량을 넘었기 때문이다. 그에 비해 BATCH와 PROPOSED는 성공적으로 수행되었다. 더욱이 |F| = 10^6일 때와 비교하여 BATCH와 PROPOSED의 성능 격차는 더욱 커졌음을 알 수 있다. 이것은 PROPOSED가 F에 비해 훨씬 작은 큐보이드들을 사용하여 다른 큐보이드들을 계산하기 때문이다. 따라서 제안 방법은 기존 방법에 비해 수행해야 하는 맵리듀스 잡의 개수는 증가하지만, 데이터 큐브를 더 빠른 시간에 계산할 수 있음을 확인할 수 있다.

Fig. 7은 PROPOSED에서 수행하는 3개의 맵리듀스 잡 각각의 수행시간을 나타낸다. 앞서 설명한 바와 같이 |F| = 10^6일 때는 |F|와 |*abcd*|의 차이가 그리 크지 않으므로, *abcd*에서 3개의 큐보이드 *abd*, *acd*, *bcd*를 계산하는 두 번째 잡(job2)의 비용이 F에서 1개의 큐보이드 *abcd*를 계산하는 첫 번째 잡(job1)의 비용보다 크다. 하지만 비용 모델을 통해 예측한 바와 같이, *acd*와 *bcd*로부터 *ac*와 *bd*를 계산하는 세 번째 잡(job3)의 비용은 F로부터 *ac*와 *bd*를 직접 계산하는 것보다 크게 줄어들게 된다. |F| = 10^6, 10^7일 때는 앞서 설명한 바와 같이 PROPOSED에서는 맵리듀스 잡이 진행될수록 F에 비해 훨씬 작은 큐보이드들을 입력으로 사용하므로, 각 맵리듀스 잡의 수행비용이 크게 줄고 있음을 볼 수 있다.

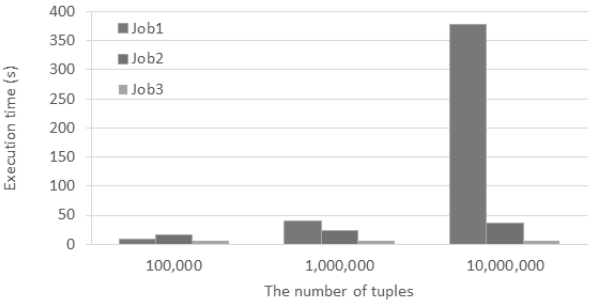

Fig. 7. Execution time of each job in the proposed method

6. 결 론

본 논문은 맵리듀스 환경에서 데이터 큐브를 효율적으로 계산하는 방법을 제안하였다. 제안 방법은 기존 방법에 비해 수행해야 하는 맵리듀스 잡의 개수는 증가하지만, 각 맵리듀스 잡에서 맵퍼가 출력으로 내보내는 중간결과의 수를 대폭 감소시킴으로써 총 수행비용을 크게 줄였다. 가상 데이터를 사용한 실험에서 제안 방법은 기존 방법에 비해 총 계산 시간을 최대 20%까지로 감소시킴을 확인하였다.

추후 연구로는 맵리듀스를 사용하지 않고 HDFS와 같은 분산 파일 시스템에 직접 접근하여 데이터 큐브를 계산하는 방법과, 맵리듀스상에서 Quotient cube[14], Condensed cube[15] 등 다른 다양한 형태의 데이터 큐브를 계산하는 방법에 대한 연구 등이 있다.

References

[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, June 27, 2011.

[3] Jeffrey Dean, Sanjay Ghemawat, “MapReduce: simplified data processing on large clusters,” In Proceedings of OSDI ’04, pp.137-150, 2004.

[4] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals,” In Proceedings of the ICDE Conference, pp.152-159, 1996.

[5] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan, “Data Cube Materialization and Mining over MapReduce,” *IEEE Transactions on Knowledge and Data Engineering*, Vol.24, No.10, pp.1747-1759, 2012.

[6] Zhengkui Wang, Yan Chu, Kian-Lee Tan, Divyakant

Agrawal, Amr EI Abbadi, and Xiaolong Xu, “Scalable Data Cube Analysis over Big Data,” CoRR, abs/1311.5663, 2013.
[7] Wang, Yuxiang, Aibo Song, and Junzhou Luo, “A mapreducemerge-based data cube construction metho,” In Proceedings of IEEE International Conference on Grid and Cooperative Computing(GCC), pp.1-6, 2010.
[8] Sergey, Kuznecov, and Kudryavcev Yury, “Applying map-reduce paradigm for parallel closed cube computation,” In Proceedings of IEEE International Conference on Advances in Databases, Knowledge, and Data Applications, pp.62-67, 2009.
[9] You, Jinguo, Jianqing Xi, and Pingjian Zhang, “A parallel algorithm for closed cube computation,” In Proceedings of IEEE/ACIS International Conference on Computer and Information Science, pp.95-99, 2008.
[10] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman, “Implementing Data Cubes Efficiently,” In Proceedings of ACM SIGMOD, pp.205-216, 1996.
[11] Kevin Beyer, Raghu Ramakrishnan, “Bottom-Up Computation of Sparse and Iceberg Cube,” In Proceedings of ACM SIGMOD, pp.359-370, 1999.
[12] Guoping Wang, Chee-yong Chan, “Multi-Query Optimization in MapReduce Framework,” *PVLDB*, Vol.7, No.3, pp.145-156, 2013.
[13] http://aws.amazon.com/ec2/
[14] Lakshmanan, Laks VS, Jian Pei, and Jiawei Han, “Quotient cube: How to summarize the semantics of a data cube,” In Proceedings of the 28th international conference on Very Large Data Bases, pp.778-789, 2002.
[15] Wang, Wei, Jianlin Feng, Hongjun Lu, and Jeffrey Xu Yu, “Condensed cube: An effective approach to reducing data cube size,” In Proceedings of IEEE International Conference on Data Engineering, pp.155-165, 2002.

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~2014년 숙명여자대학교 컴퓨터과학부 조교수
2014년~현 재 숙명여자대학교 컴퓨터과학부 부교수
관심분야 : 데이터베이스, 질의처리, 빅데이터, 데이터웨어하우스

박 소 정
e-mail : sojeongpark@sookmyung.ac.kr
2011년~현 재 숙명여자대학교 컴퓨터과학부 학사과정
관심분야 : 데이터베이스, 빅데이터

박 은 주
e-mail : peunju92@sookmyung.ac.kr
2014년 숙명여자대학교 컴퓨터과학부(학사)
2014년~현 재 숙명여자대학교 컴퓨터과학부 석사과정
관심분야 : 데이터베이스, 빅데이터

박 진 경
e-mail : jinkyung93@sookmyung.ac.kr
2012년~현 재 숙명여자대학교 컴퓨터과학부 학사과정
관심분야 : 데이터베이스, 빅데이터

최 연 정
e-mail : cyj@sookmyung.ac.kr
2012년~현 재 숙명여자대학교 컴퓨터과학부 학사과정
관심분야 : 데이터베이스, 빅데이터