데이타웨어하우스 환경에서의 질의 처리 성능 향상을 위한 캐시 관리자

(A Cache Manager for Enhancing the Performance of Query Evaluation in Data Warehousing Environment)

심 준 호 †

(Junho Shim)

요 약 데이타웨어하우스는 의사결정시스템의 질의처리에 사용되는데, 통상적으로 의사결정질의의 응답 속도는 OLTP 질의 응답속도에 비해 수십 배 이상 오래 걸린다. 의사결정은 대부분 빠른 시간 안에 이루어지는 것이 필수적이므로 의사결정질의 응답 속도를 단축시키는 기술은 중요하다. 본 논문에서는 기존의 질의결과를 캐싱하여 주어진 질의처리에 이용하는 기법을 제시한다. 이를 위해 먼저 의사결정시스템이 이 기법에 적합한 환경을 가지고 있는지 살펴본다. 그다음, 임의 형태의 모든 질의를 처리한다는 것은 불가능하므로 우리가 다루는 질의 형태인 정규화형태를 정의한다. 질의가 정규화형태를 따르지 않으면 단순 스트링 매칭을 하고, 정규화된 경우라면 질의스플릿이란 질의 변환 과정과 질의종속그래프를 통해 캐시된 질의결과를 찾은 후 그 결과위에서 질의를 수행한다. 캐시 관리자는 질의응답시간을 최소화하도록 캐시를 유지해야한다. 이를 위해 질의 수행비용, 질의결과의 크기, 레퍼런스비율, 베이스 테이블의 업데이트비율 및 그에 따른 질의결과 유지비용 등을 고려하여 캐싱하는 동적 캐시교환기법을 제안한다. 제안된 기법은 실험을 통해 그 성능을 검증하였다.

키워드 : 데이타웨어하우스, 동적 캐싱, 데이타큐브, 의사결정질의

Abstract Data warehouses are usually dedicated to the processing of queries issued by decision support system (DSS). The response time of DSS queries is typically several orders of magnitude higher than the one of OLTP queries. Since DSS queries are often submitted interactively, techniques for reducing their response time are important. The caching of query results is one such technique particularly well suited to the DSS environment. In this paper, we present a cache manager for such an environment. Specifically, we define a canonical form of query. The cache manager looks up a query based on the exact query match or using a suggested query split process if the query is found in non-canonical form or in canonical form, respectively. It dynamically maintains the cache content by employing a profit function which reflects in an integrated manner the query execution cost, the size of query result, the reference rate, the maintenance cost of each result due to updates of their base tables, and the frequency of such updates. We performed the experimental evaluation and it positively shows the performance benefit of our cache manager.

Key words : data warehouse, dynamic caching, data cube, decision support system query

1. 서 론

데이타웨어하우스(data warehouse)는 흔히 다수의 이질적인(heterogeneous) 데이타베이스로부터 필요한 정보를 통합한 하나의 데이타보관소(data repository)이다[1]. 데이타웨어하우스는 주로 의사결정시스템(decision support system)과 데이타마이닝(data mining)과 같은 자료 분석시스템에 사용되는데, 시간이 지남에 따라 원하는(interesting) 데이타가 많아지게 되므로 웨어하우스의 크기 역시 커지게 된다[2]. 몇 개의 튜플만 액세스하는 기존의 OLTP(OnLine Transaction Processing)와 달리 의사결정시스템이나 데이타마이닝에서의 질의(query)들은 그 처리가 매우 복잡하고 웨어하우스에

† 정 회 원 : 숙명여자대학교 정보과학부 교수
jshim@sookmyung.ac.kr
논문접수 : 2002년 11월 20일
심사완료 : 2003년 3월 29일

저장되어 있는 데이타의 상당부분을 액세스해야만 하므로 그 처리속도가 OLTP 질의 처리 속도에 비하면 수십 배에서 수천 배(several orders of magnitude) 정도 더 걸리게 된다. 하지만 의사결정시스템 질의는 용어 그대로 의사 결정에 필요한 정보를 제공해 주는 것이 그 목적이므로 많은 경우에 있어 질의 처리가 신속하게 이루어져야지 너무 느리게 이루어진다면 그 효용이 크게 떨어지게 된다[1,2].

따라서 데이타웨어하우스에서의 질의 처리 속도를 높이기 위한 다각적 연구가 이루어져 왔는데, 그 중의 한 방법이 과거의 질의결과를 케싱(caching)하여 나중의 질의 처리에 이용하는 기법이다[3,4]. 데이타웨어하우스 환경의 다음과 같은 세 가지 특성은 질의결과케싱이 질의처리 성능향상에 도움이 될 수 있는 좋은 환경을 제공한다[1,5].

1) OLTP 시스템과 달리 데이타웨어하우스는 상대적으로 업데이트가 훨씬 덜 빈번하게 일어난다.
2) 데이타웨어하우스에서의 상당수 의사결정질의는 그 처리 시간은 오래 걸리지만 결과 값은 저장 공간을 그리 많이 차지 않는다.
3) 데이타웨어하우스에서의 의사결정질의들은 대개 "드릴다운(drill-down)"/"롤업(roll-up)" 형식 등의 패턴을 따르며 그러한 의사결정질의들은 상당수 일정한 형태로 표현되어진다.

하지만 과거의 질의결과가 미래의 질의 처리에 효율적으로 이용될 수 있기 위해서는 풀어야 할 많은 문제들이 있는데, 다른 문제들 못지않게 중요한 것들로서 다음의 두 가지를 들 수 있다.

1) 한정된 케싱 저장 공간에서의 질의결과캐시교환(cache replacement)
2) 효율적 질의포함관계(query subsumption) 체크

본 연구는 위 두 가지 문제들에 대해 합리적이고 효율적인(reasonably efficient) 방향 제시 없이는 데이타웨어하우스 상의 질의결과케싱이 그 효용성에 한계가 있으므로, 위의 문제 해결을 위해 부분별로 행해진 기초연구[4,5]를 바탕으로 본 논문에서는 기존의 연구를 하나의 통합된 환경으로 체계적인 확장을 한다.

이를 위해 본 논문에서는 먼저 일정한 형태로 표현되는 의사결정질의의 형태를 찾아, 그 형태를 *정규화형태(canonical form)*로 정의하고, 정규화형태 질의의 효율적인 질의포함관계를 위한 *질의스플릿(query split)* 알고리즘과 *질의종속그래프(query attachment graph)*를 제시한다. 정규화형태를 따르지 않는 질의들은 질의포함관계를 스트링매칭과 같은 단순매칭(exact matching)을 사용한다.

유한 공간 캐시에서 새로운 질의결과를 위한 공간이 부족한 경우, 새로운 질의결과는 저장된 기존의 질의결과들 가운데 제일 이득이 없을 것이라 판단되는 것과 교환(replace)하는 동적 질의결과캐시교환 기법을 제시한다. 제시된 기법은 질의의 수행비용, 질의결과의 크기, 질의결과의 레퍼런스비율(reference rate)과 베이스 테이블의 업데이트비율과 업데이트에 따른 질의결과 유지비용등을 *이득(profit)*이라는 통합된 식으로 계산해 각 질의결과의 캐싱 여부를 고려하게 된다. 기법은 저장될 질의결과 집합을 미리 결정하는 정적(static) 방식이 아닌, 질의패턴변화 등의 웨어하우스 환경 변화에 적응할 수 있는 동적(dynamic) 방식이다. 질의결과캐시교환 기법은 질의종속그래프로 표현되는 질의포함관계를 반영하도록 이득 식을 확장시킨다.

제시된 알고리즘을 이용한 캐싱 기법은 실험을 통해 성능평가를 하였으며, 실험 결과는 본 논문에서 제시하는 캐시 관리자가 질의 처리 성능을 향상시킴을 보이고 있다.

이후의 본 논문 구조는 다음과 같다. 먼저 2절에서는 관련 연구를 보이고, 3절에서는 질의 스플릿 과정과 질의종속그래프를 소개한다. 4절에서는 동적 질의결과캐시교환 기법을 소개한다. 5절에서는 본 논문에서 제시한 캐싱기법의 성능을 실험을 통해 보이고, 마지막으로 6절에서는 결론을 유도한다.

2. 관련 연구

캐시 관리자의 성능은 어떤 캐시교환 기법을 쓰는지에 따라 상당히 달라질 수 있다. 따라서 본 논문의 기초가 되는 초기 연구는 주로 질의결과의 효율적인 캐시교환 알고리즘에 집중하였다[4]. 캐시교환 기법은 운영 시스템이나 데이타베이스 버퍼링 분야 등에서 심도 있게 연구되었다[6]. 대개의 캐시교환 기법은 가장 높은 레퍼런스 빈도를 보이는 페이지를 캐시하는, 이른바 힛-레이쇼(hit ratio)를 극대화하는데 주력한다. 하지만 [4]에서처럼, 데이타웨어하우스 환경에서는 각 질의결과의 크기, 질의의 평균 레퍼런스비율, 질의 수행비용 등을 고려한 이득(profit)에 따른 캐싱이 훨씬 효과적임이 알려져 있다.

과거의 질의결과를 캐싱해놓고 미래의 질의 수행에 캐싱결과를 이용하는 기법은 [7,8,9] 등에서 연구되었다. 그룹핑이나 애그리게이션(aggregation)이 없는 제한된 형태의 질의포함관계는 [7,8]에서 연구되었다. 뷰를 이용한 애그리게이션 질의에 대한 연산 기법은 [9,10]에서

연구되었는데, [9]에서는 뷰 정의가 질의 정의와 일치하는지를 가리기 위한 질의 구문 변환 기법을 소개하고 있다. [10]에서는 고정된 크기의 청크(chunk)를 캐싱하여 질의 연산을 시도하는데, 청크의 크기 등을 어떻게 잡는지가 이러한 캐싱기법의 성공여부를 결정하게 된다.

데이타큐브의 선택적인 캐싱은 [3,11] 등에서 연구되었다. [3]은 롤업, 드릴다운등이 자주 사용되는 OLAP 환경에서 SQL group by 연산의 문제를 제기하고, cube by 연산자를 소개한다. 또 데이타큐브에 사용되는 다양한 애그리게이트 함수들을 카테고라이즈 해놓고 있다. 이들 연구는 질의수행시간 최소화를 목표로 benefit 이라는 함수값에 이용해 그리디(greedy) 방식으로 캐싱할 데이타큐브 집합을 정한다. 선택은 모두 정적으로 이루어져 일단 선택된 큐브들의 집합은 질의 패턴의 변화에 따라 변경되지 않는다. 본 논문의 캐시관리자는 질의 환경 변화에 따른 동적 캐싱이란 면에서 구분된다. [3]은 본 논문이 제시하는 알고리즘과 성능평가의 대상이 된다.

마지막으로 본 논문과 본 논문의 부분별 선행 연구인 [4,5]와의 관계는 다음과 같다. 먼저 [4]에선 주로 한정된 캐싱 저장 공간에서의 질의결과의 효율적 선택을 위한 질의결과캐시교환 기법의 제시를 하였다. [5]에서는 데이타웨어하우스에서의 질의간의 포함 관계를 캐싱에 도입하였다. 본 논문에서는 [4]의 캐시교환 알고리즘에서 제시된 목적함수와 이득함수 등이 질의간의 포함 관계를 고려하는 캐시교환에서도 어떻게 확장 사용될 수 있는지를 상세하게 보인다. 정규화형태의 질의 구문 특성과 질의포함관계를 위한 질의종속그래프의 효율적인 자료구조와 알고리즘도 제시한다. 마지막으로 질의가 정규화형태를 따르는 경우와 그렇지 못한 경우를 체크하여 하나의 통합 환경에서 캐싱해주는 알고리즘을 제시한다. 제시된 알고리즘의 오버헤드와 이득함수에서의 캐시유지비용 요소의 역할 등이 실험을 통해 분석된다. 이러한 점들은 [4,5]에서는 보이지 못한 본 논문만이 가지는 공헌이라 할 수 있다.

3. 질의 스플릿 과정과 질의 종속 그래프

3.1 배경 설명과 예

캐싱 공간에 결과가 미리 저장된 질의가 임의(arbitrary)의 형태를 띤 새로운 질의와 포함(subsumption) 관계가 있는지를 구하는 문제는 NP 문제로 알려져 있다. 하지만 제한되어진 특수한 환경에서는 그 질의 처리가 상대적으로 효율적으로 이루어질 수도 있는데, 이러한 환경에서 질의들은 미리 예측 가능하다거나 일정한 어떤 형태를 취하고 있는 경우가 대부분이다[7,8]. 데이타웨어하우스에서의 데이타 스키마는 통상적으로 스타 스키마(star schema)와 같은 표준적 스키마를 따르고, 이에 질의되는 의사결정질의들 역시 상당수 일정한 형태로서 표현될 수 있다는 사실은 데이타웨어하우스에서의 질의포함관계 분석이 어느 정도 효율적으로 될 수 있다는 가능성을 열어준다[5,6]. 예를 들어 의사결정시스템 벤치마킹도구로서 자주 이용되는 TPC-H/R에서의

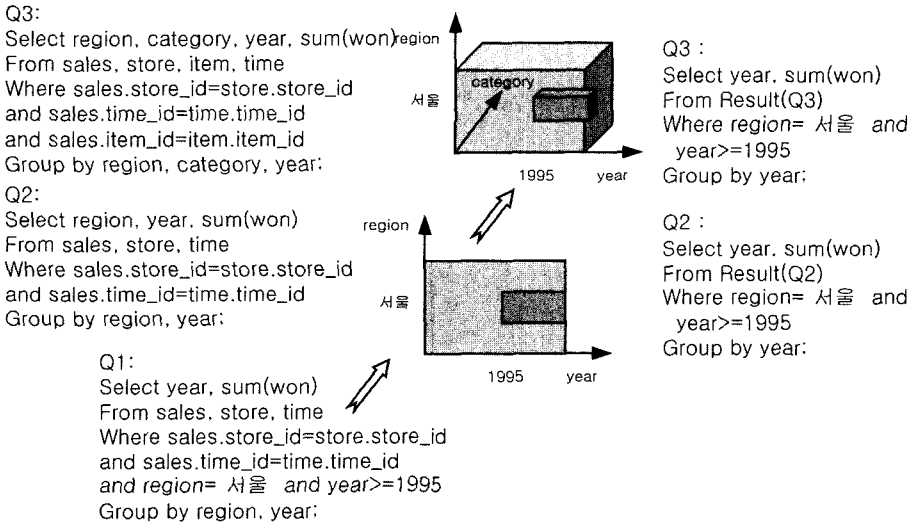

그림 1 정규화형태 질의와 질의 수행

질의들의 상당수가 일정한 형태를 따르고 있다[12].

예를들어 할인매장의 대표이사가 대리점들에 대한 인원 및 투자계획에 대한 증감유무의 결정을 내리기 위해, 운영해오던 데이타웨어하우스에 그림 1에서 보이는 바와 같이 "서울 지역 대리점들의 1995년 이후의 총 판매액을 구하시오"란 SQL 질의 Q1을 하였다 하자. 캐시에 Q2 질의처리 결과가 저장되어있다면 Q2' 이란 변형된(transform)된 질의를 Q2의 결과 위에서 수행함으로서 Q1의 결과 값을 구할 수 있다. 만약 Q2의 결과가 캐시에는 없지만 Q3의 질의결과가 캐시에 있다면 비슷한 방식으로 Q3' 이란 변형된 질의를 Q3의 결과 위에서 수행함으로서 그 결과 값을 구할 수 있다. Q1과 같은 형태의 질의가 바로 *정규화형태(canonical form)*를 띈 질의의 예이며, 정규화형태 질의는 Q2와 Q2', Q3와 Q3'과 같은 질의들로 시스템적으로 변형되어 그 처리를 캐싱된 다른 질의처리 결과 위에서 할 수 있는지 효율적으로 검색할 수 있다.

3.2 정규화형태 질의와 질의스플릿 과정

질의 Q가 다음과 같은 형태를 따르고 있다면 Q는 *정규화형태(canonical form)*를 따른다 한다.

```
select select-list, aggregate-list
from t1,t2,....,tn
where join-condition AND select-condition
group by group-by-list
having group-condition; (이절은 옵션절이다.)
```

여기서 *t1,t2,...,tn*은 테이블 리스트이고, *join condition*은 *t1,t2,...,tn*을 조인하기 위한 AND로 연결된 절이다. *select-condition*은 프레디킷(predicate)의 불린(boolean)조합 절인데, 각 프레디킷은 하나의 속성(attribute)만 가지고 {<,>,≤,≥,=,≠,between}에 속하는 영역 비교(range comparison) 연산자를 가지는 제한이 있다.

예를 들어 앞 절의 Q1은 다음과 같이 정규화형태를 따른다. select-condition의 각 프레디킷이 하나의 속성만 가진다는 의미는 region="서울" 처럼 비교 연산자의 한쪽에만 속성이 나타난다는 의미이다.

```
select-list: year, aggregate-list: sum(won)
t1: sales, t2: store, t3: time
join-condition: sales.store_id=store.store_id AND
sales.time_id=time.time_id
select-condition: region="서울" and year>=1995
group-by-list: region, year
```

select condition과 group condition이 널인 정규화형태 질의를 *데이타큐브질의*라 한다[3]. 다시 말해 데이타큐브질의는 질의결과가 *데이타큐브*인, 정규화형태 질의의 한 특수한 형태인 것이다. 데이타큐브는 유도관계(derivability relation)에 있어 래티스(lattice) 구조를 따르고 래티스에서 링크는 데이타큐브간의 *parent, child* 관계를 나타낸다. 어떤 데이타큐브 c의 *IA(immediate alive ancestor)*란, 래티스 구조에서 c의 parent 관계로 나타나는 조상(ancestor)큐브 중에서 그 결과가 캐시에 저장되어 있는 첫 번째 것을 지칭한다.

질의의 select-condition에 나타나는 속성들에 대해 애그리게이션 함수값을 가진 데이타큐브가 캐시에 있다면, 그 데이타큐브는 주어진 질의의 수행에 이용될 수 있다. 만일 해당 데이타큐브가 캐시에 없다면 래티스 구조를 따라 조상큐브를 따라가면서 어떤 조상 큐브가 IA가 되는지를 살펴, IA를 질의 연산에 이용할 수 있다. 이 과정이 질의스플릿 과정의 아이디어가 된다.

그림 2에서 Q⇒(Q1, Q2)는 정규 형태를 따르는 Q가 다음과 같은 두 개의 질의 Q1, Q2로 바뀌는 과정을 의미한다.

```
Q ⇒ (Q1, Q2)
base query ← Q1, slice query ← Q2
i ← 0, Base_i ←{Q1}
While no base query in Base_i is materialized in the cache
begin
  i ← i + 1;
  for each Q ∈ Base_i begin
    Base_i ← Base_i ∪ {Q'} where Q' ∈ parents(Q)
  end for
end while
if Base_i = φ begin
  IA ← raw data
  slice query ← Q
end if
else begin
  Choose b ∈ Base_i s.t. b is the data cube with the least
estimated cost to evaluate Q using it
  IA ← b
  slice query is the rewritten query Q to use b
end else
return base query, IA, and slice query
```

그림 2 질의스플릿 알고리즘

```
Q1: select select-list1,
    aggregate-list
    from t1,t2,....,tn
    where join-condition
    group by group-by-list1

Q2: select select-list,
    aggregate-list
    from RS1
    where select-condition
    group by group-by-list
```

여기서 *select list1*과 *group-by-list1*은 select-condition에 나타나는 각 컬럼 이름에 일대일(one-to

one)로 매핑(mapping) 리졸브(resolve)되는 컬럼만으로 구성된 리스트이고, *RS1*이란 Q1의 질의결과(result set)를 의미한다. 이 경우 Q1은 Q의 *베이스질의(base query)*라고 부르고, Q2는 *슬라이스질의(slice query)*라 부른다. 베이스질의의 결과가 캐시에 저장되지 않은 경우에는, 베이스 질의의 parent 중 캐시에 결과가 있는 IA를 찾고 Q1과 Q2는 IA를 이용하도록 다시 쓰이게 된다. 참고로 베이스질의의 형태는 데이타큐브 질의임을 유의하기 바란다.

3.3 질의종속그래프

정규화형태 질의 Q가 질의스플릿 과정을 거쳐 베이스질의나 IA, 슬라이스질의로 변형되면, 질의 Q 연산결과를 베이스질의나 IA 결과위에서 슬라이스질의를 수행함으로서 얻을 수 있다. 각 질의간의 베이스 슬라이스 관계는, 즉 질의포함 관계는 그림 3과 같은 질의종속그래프를 통해 캐시 관리자의 메타 저장 영역에서 관리된다.

그래프에서 질의 옆의 숫자는 질의결과의 크기를 의미하며, 링크(→:Ed 링크로 표시, edge for derivability)는 질의스플릿 과정에서 설명한 질의간의 베이스 슬라이스 관계를, 그리고 링크위의 숫자는 질의의 베이스질의 결과위에서의 수행비용을 나타낸다. 다른 종류의 링크(↔:Ee 링크로 표시, edge for equality)는 인접한(incident) 두 개의 질의가 동일결과값 질의(result equivalent queries)를 의미한다. 예로서 주어진 질의 Q가 질의스플릿 과정을 통해 Q⇒(Q4, Q3′)로 되었는데, Q4 결과가 캐시되지 않았으므로 IA가 Q5로 되고 추가적인 변환을 거쳐 (Q5, Q3″)이 된다. 이때 Q3′과 Q3″은 동일결과값 질의이며 실제 Q의 수행은 Q5의 결과값 위에서 Q3″을 수행함으로 이루어지게 된다.

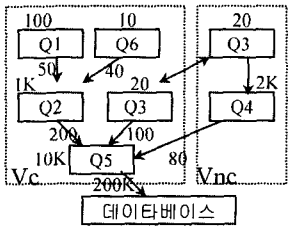

그림 3 질의종속그래프의 예

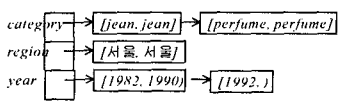

그림 4 질의포함관계를 위한 해쉬구조

각 노드에는 질의 구문정보와 캐시교환 알고리즘에서 사용될 각 질의결과의 이득(profit) 계산에 필요한 정보들이 저장되어진다. 시간이 지남에 따른 너무 많은 노드로 인한 그래프 관리의 오버헤드는 정기적으로 가장 다지막으로 캐싱된 질의 노드의 이득보다 이득이 적은 모든 질의를 제거(garbage collection)함으로서 해결할 수 있다. Vc와 Vnc는 각각 그 영역에 나타나는 노드들이 질의결과가 캐싱되었는지 아닌지를 표현한다. 현재 캐시되지 않은 질의 노드도 관리하는 이유는 나중에 해당 질의의 레퍼런스 빈도수가 높아짐과 같은 환경의 변화에 따라 이득값이 높아질 수 있는데, 이 경우 그 질의결과를 캐싱하기 위해서이다. 다시 말해 정적 캐싱이 아닌 동적 캐싱을 하기위해서는 설사 결과가 현재 캐시되지 않았다해도 이득 계산에 필요한 정보가 저장 관리되는 것이다.

질의종속그래프의 각 노드는 질의의 select-condition에 나타나는 속성에 대한 영역 값을 각 *인터벌트리(interval tree)*로 가지는 해쉬테이블을 가진다[13]. 예를 들어 한 정규화형태 질의가 select-condition으로 ((category="jean" or category="perfume") and region="서울" and ((year>=1982 and year<1990) or (year >=1992)))를 가진다면 이 질의 노드에 있는 해쉬테이블 구조는 그림 4와 같다. 그림에서 (혹은)은 인터벌이 열려있음을, [혹은]는 인터벌이 닫혀있음을 의미한다. 각 질의에 대해 이러한 해쉬테이블을 구성하기 위한 시간복잡도(time complexity)는, l이 하나의 속성에 나타나는 최대 프레디킷 개수이고 m이 select-condition에 나타나는 서로 다른 속성 개수인 경우, 소팅하는데 $O(m \cdot l \cdot log\ l)$, 인터벌트리를 구성하는데 $O(l \cdot log\ l)$의 복잡도를 가진다[13].

주어진 질의가 질의종속그래프에 나타나는 어떤 질의에 포함(subsume) 되는지는, 주어진 질의의 베이스질의나 IA에 연결된 각 질의 노드의 해쉬테이블을 따라 질의의 select-condition에 나타나는 각 속성의 인터벌을 모두 포함하는지의 여부를 따라 찾을 수 있게 된다. 따라서 질의포함관계 체크를 위한 복잡도는 n이 주어진 질의의 베이스 질의나 IA에 연결된 최대 질의 개수를 나타낼 때 $O(n \cdot m \cdot l \cdot log\ l)$로서 지수함수가 아닌 다항식값(polynomial time)안에 이루어질 수 있다.

4. 동적 질의결과캐싱교환 기법

이 절에서는 제한된 캐시 공간에서 질의결과들이 어떠한 선택 과정을 거쳐 선별적으로 캐시에 저장 보관되는지를 살펴본다. 먼저 4.1절에서는 임의의 질의를 수행

하는데 그 질의결과가 캐싱된 경우에는 캐시로부터 그 결과를 가져오고, 캐시되지 않은 경우에는 데이타베이스에서 직접 수행하는 기본 모델에 따른 질의결과캐싱교환 기법(cache replacement algorithm)을 설명한다. 만일 정규화 형태 질의처럼, 주어진 질의의 패턴을 분석하여 질의 수행이 해당 질의가 아닌 캐시된 다른 질의결과를 이용한다면, 기본 모델은 이에 따라 확장되어야 하고 질의결과캐싱교환 기법 역시 확장되어져야 한다. 이는 4.2절에서 설명하기로 한다. 기본 모델과 이의 확장 모델에서 설명된 질의결과캐싱교환 기법들은 확장 모델에서의 질의결과캐싱교환 기법으로 통합될 수 있는데 이에 대한 몇 가지 사항을 4.3절에서 설명한다.

4.1 기본 모델

전체 질의수행 비용을 최소화하도록 캐싱을 하는 것은 질의응답시간을 최소화하는 한 방법이 될 수 있다. 이를 포말(formal)하게 살펴보자. Q를 질의 qi의 집합이라 하고, RS를 캐시된 질의결과 RS_j의 집합이라 하자. I와 $J(J \subseteq I \subseteq N$, N은 자연수집합)는 각각 $qi \in Q$ iff $i \in I$, $RSj \subset RS$ iff $j \in J$ 인 인덱스 집합이다. ci는 qi가 데이타베이스로부터 수행되는 비용(cost)이고, ri는 qi가 나타나는 레퍼런스비율(reference rate)이고, si는 qi의 결과인 RS_i의 크기이고, S는 캐시 공간의 크기이다. 질의결과가 캐시된경우 캐시로부터 그 질의결과를 가져오는 비용은 무시할 수 있다하자. 그러면 질의결과캐싱교환 알고리즘은 캐시된 전체 질의결과 크기는 캐시공간보다 작으면서도, 캐싱에 힛(hit)하지 않는 질의 수행에 필요한 전체 비용을 최소화하도록 다음과 같은 목적함수(objective function)를 최소화해야 한다.

$$\min \{ \sum_{k \in I} (r_k \cdot c_k) \} \text{ with subject to } \sum_{k \in J} s_k \leq S \quad (1)$$

이 문제는 Knapsack 문제로 변환(reduce)될 수 있으므로 NP-complete 문제이다[7]. 따라서 휴리스틱을 써야 하는데, 우리는 캐시안에 저장되는 질의결과들의 수행비용, 레퍼런스비율, 질의결과 크기 등을 모두 고려해, 캐시함으로서 얻는 이득이 제일 큰 순서대로 정렬한 후 저장된 공간이 허용하는 만큼의 질의결과만을 저장하는 방식의 그리디 휴리스틱(greedy heuristic)을 취한다. 질의 q_i의 수행이 데이타베이스로부터 이루어지지 않고 캐시로부터 그 결과를 가져오는 빈도수 (다시 말해 q_i가 질의되었을 때 해당 결과 RS_i가 캐시에 있는 경우의 빈도수)를 h_i라 하면, 질의응답시간의 최소화는 아래와 같은 *비용절감비율(CSR: cost saving ratio)*을 최대한으로 하는 것이 바람직할 것이다.

$$CSR = \frac{\sum_{j \in J} c_j \cdot h_j}{\sum_{i \in I} c_i \cdot r_i} \quad (2)$$

CSR을 최대한으로 하기위하여 캐시관리자의 질의결과캐시교환 기법은 각 질의결과 아이템별로 그 질의결과를 캐싱함으로부터 얻는 비용절감 효과가 크도록 하는게 바람직한데, 이를 위해 *이득(profit)* 이라는 통일된 척도를 가지고 각 아이템별로 이득함수 값을 계산하여 가능한 그 값이 큰 아이템을 캐시내에 저장하려고 하는 것이다.

$$profit(RS_i) = \frac{\lambda_i \cdot c_i}{s_i} \quad (3)$$

여기서 λ_i는 q_i의 평균레퍼런스비율(average reference rate)으로서, LRU-K 알고리즘 방식에 의한, RS_i에 대한 최근 K 인터-어라이벌(inter-arrival) 시간에 대한 이동평균(moving average) 방식으로 측정한다[6].

$$\lambda_i = \frac{K}{t - t_K} \quad (4)$$

여기서 t는 현재 시간이고 tk는 현재부터 과거로 지난 K번째 레퍼런스 시간이다. RS_i에 대한 최근 레퍼런스 시간이 K개가 되지 않는 경우에는 가용한 최대 개수를 사용하여 λ_i를 구하지만, λ_i의 정확한 계산에 대한 신뢰도가 떨어지므로 K개가 안되는 질의결과는 질의교환시 K개를 모두 가진 동일한 조건의 질의결과에 비해 캐싱되는 우선순위가 처지게 함이 바람직하다. 또 정기적으로 슈도레퍼런스에이징(psuedo reference aging)을 함으로서 장시간 레퍼런스는 되지 않았더라도 상대적으로 높은 λ_i를 야기하는 현상을 방지한다. 마지막으로 이득함수가 질의결과 크기(s_i)에 대해 나누어진 것은 주어진 캐시 공간 안에서 $\lambda_i \cdot c_i$뿐만 아니라 크기를 고려하도록 그 값을 정규화(normalized)함이 바람직해서이다.

4.2 정규화형태 질의간 포함관계를 고려한 확장 모델

주어진 질의의 패턴을 분석하여 질의 수행이 해당 질의가 아닌 캐시된 다른 질의결과를 이용하는 경우 질의응답시간을 최소화하려면, 앞 절의 기본 모델에서처럼 질의수행에 필요한 비용을 최소화함과 아울러, 데이타베이스의 업데이트에 따른 캐시 질의결과의 유지비용(maintenance cost)도 고려해야한다. 이를 포말하게 보이기 위해 앞 절의 기본 모델에서 사용한 목적함수와 최적화를 위해 사용한 비용절감비율(CSR)과 이득(profit) 함수를 확장한다.

Q를 질의 q_i의 집합이라 하고, RS를 캐시된 질의결과 RS_i의 집합이라 하자. I와 $J(J \subseteq I \subseteq N$, N은 자연수

집합)는 각각 $q_i \in Q$ *iff* $i \in I$, $RS_j \in RS$ *iff* $j \in J$인 인덱스 집합이다. c_{ij}는 q_i가 RS_j를 이용해 수행된 경우의 비용이고, c_i (즉 $c_{i,database}$)는 q_i가 데이타베이스로부터 수행되는 비용이며, r_{ij}는 q_i가 RS_j를 이용해 수행된 경우의 레퍼런스비율이다. u_j는 데이타베이스가 업데이트 됨에 따라 필요한 RS_j의 업데이트비율(update rate)이고, c_j^*는 RS_j의 업데이트비용이다. q_i는 $RSj \in RS$인 모든 j 에 대해 $c_i < c_{ij}$인 경우 데이타베이스로부터 수행될 것이다. s_i는 q_i의 결과인 RS_i의 크기이고, S는 캐시 공간의 크기이다. 그러면 질의결과캐시교환 알고리즘은 캐시된 전체 질의결과 크기는 캐시공간보다 작으면서도, 질의 수행에 필요한 총 비용과 동시에 캐시된 질의결과를 유지하는데 필요한 총 비용을 최소화하도록 해야하는데, 다음과 같은 목적함수로 나타낼 수 있다.

$$\min \{ \sum_{i,j}(r_{ij} \cdot c_{ij}) + \sum_{j}(u_j \cdot c_j^*) \} \text{ with subject to } \sum_{j} s_j \le S \quad (5)$$

앞 절의 기본 모델에서는 캐시된 질의결과의 업데이트를 하지 않으므로 모든 u_j는 0이고, q_i의 캐시 힛일때의 비용을 무시하고 오로지 캐시 미스(miss)일 경우의 데이타베이스로부터의 $c_i(c_{i,database})$ 만을 고려하였므로, 위의 목적 함수는 이전의 목적 함수 식 (2)가 확장된 것임을 알 수 있다. 이 목적 함수를 최소화하기 위한 그리디 휴리스틱 역시 일관된 방법으로 확장될 수 있다.

질의 q_i의 빈도수가 r_i이고, h_{ij}는 q_i가 RS_j를 이용해 수행된 빈도수라 하면, 질의 응답 시간의 최소화는 기본 모델의 비용절감비율(CSR)을 다음과 같이 확장시킨 값을 최대한으로 해야한다.

$$CSR = \frac{\sum_{i \in I, j \in J}(c_i \cdot r_i - c_{ij} \cdot h_{ij} - c_j^* \cdot u_i)}{\sum_{i \in I} c_i \cdot r_i} \quad (6)$$

질의스플릿과 캐시 룩업(lookup)의 비용이 질의수행 비용에 비해 무시할만(negligible)하다면, 위의 CSR은 질의수행 비용절감($c_i \cdot r_i - c_{ij} \cdot h_{ij}$)에서 캐시 유지비용 ($c_j^* \cdot u_i$)을 뺀, 실제 캐시 관리자에 의한 비용 이득을 나타내게 된다. 기본 모델에서는 주어진 질의와 다른 질의의 포함관계를 따지지 않은 질의 자체의 힛 혹은 미스만을 고려했으므로 c_{ij}, h_{ij}, c_j^*, u_i 등은 모두 0으로 고려하지 않았다.

위의 CSR을 최대화하기 위해 질의 q_i의 결과아이템 RS_j의 이득(profit) 함수는 아래와 같이 확장된다.

$$profit(RS_j) = \frac{\sum_{i \in I, k \in K}(\lambda_i \cdot benefit_{ijk}) - \mu_j \cdot c_j^*}{s_j}$$

여기서 I와 K는 $q_i \rightarrow q_j$, $q_j \rightarrow q_k$인 i, k의 집합이고, μ_j는 RS_j의 평균 업데이트비율을 나타낸다.

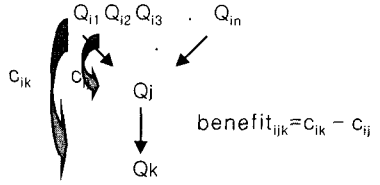

그림 5 benefitijk: cik - cij

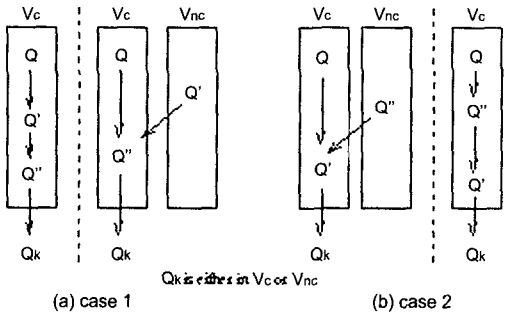

그림 6 이득함수 값의 변화

그리고 $benefit_{ijk}$는 q_i를 RS_k 대신 RS_j를 이용하여 수행하는 경우의 수행비용의 이득, 다시 말해 $c_{ik} - c_{ij}$를 의미한다. i≠j이고 $q_j \rightarrow q_k$인 q_i가 존재하지 않는다면, 즉 $i=j$이면, $benefit_{jjk}$는 $c_{jj} = 0$이므로 q_j를 RS_k를 이용해 수행하는 비용이 된다. 만일 c_{ik}의 정확한 값을 알지 못한다면, 간단한 선형비용모델(linear cost model)을 사용하여 $c_{ik} = c_{ij} \cdot \frac{|RS_k|}{|RS_j|}$ 로 추정할 수 있다. μ_j는 λ_i(평균레퍼런스비율)처럼, 최근 K 인터-어라이벌시간에 대한 업데이트 이동평균방식으로 측정한다.

4.3 이단계 동적 질의결과캐싱교환 알고리즘

- 이득함수의 동적 계산과 노드 헤더

질의 Q의 수행비용과 benefit 은 캐시의 현재 상태에 의존한다. 질의 Q의 베이스 질의나 IA 가 Q''으로 변하면, Q의 이득함수값도 변하게 된다. 이러한 경우는 Q'의 질의결과가 캐시되지 않는 경우, 즉 Q'가 Vc(캐시된 질의 노드 집합)에서 Vnc(캐시되지 않은 질의 노드 집합)으로 이동할 경우나(그림 6의 case 1), Q''이 Vc에 새로이 들어올 경우(그림 6의 case 2)이다. 케이스 1의 경우에는 수행비용과 benefit 은 다음과 같이 변경되어지게 된다.

$c_{QQ''} = c_{QQ'} \cdot \frac{|RS_{Q''}|}{|RS_{Q'}|}$, $benefit_{QQ''} = c_{QQk} - c_{QQ''} = c_{QQ''} \cdot \frac{|RS_k|}{|RS_{Q''}|} - c_{QQ''}$

케이스 2의 경우에는 수행비용과 benefit은 다음과 같이 변경되어지게 된다.

$c_{(Q)Q''} = c_{(Q)Q'} \cdot \frac{|RS_{Q'}|}{|RS_{Q}|}$, $benefit_{(Q)Q''} = c_{(Q)Q_k} - c_{(Q)Q''}$

$c_{Q'Q'}$는 불변, $benefit_{Q'(Q')} = c_{(Q')Q_k} - c_{(Q')Q'} = c_{(Q')Q'} \cdot \frac{|RS_{Q_k}|}{|RS_{Q'}|} - c_{Q'Q'}$

이득함수 값을 동적으로 계산하기 위하여, 질의종속그래프의 각 노드는 표 1과 같은 구조를 가진다.

*질의id*는 각 질의에 부여된 유일한 id이고, *정규화형태*는 질의 포함관계를 체크하는데 쓰이는 질의의 정규화된 형태이고 질의가 정규화형태를 따르지 않은 경우는 단순 스트링 매칭을 위해 질의 전체가 스트링형태로 표현된다. *타입*은 질의가 정규화형태를 따르는지 아닌지를 나타내는 태그이며, *레퍼런스리스트*와 *업데이트리스트*는 각각 K개의 최근 레퍼런스 시간과 업데이트 시간이 기록된다. *결과크기*는 질의결과의 바이트 수이며, 노드에는 실제 결과가 아닌 결과에 대한 포인터를 저장하고, 이 결과를 이용해 수행될 수 있는 질의들의 id 리스트는 *링크된 자손 리스트 필드*에 기록된다.

- 이단계 동적 캐싱관리자

현재까지 설명한 정규화형태 질의, 질의스플릿과정, 질의종속그래프, 동적 질의캐싱교환 알고리즘 등을 사용한 캐시관리자의 슈도(pseudo) 코드로 보여주면 전 페이지의 그림 7과 같다. 캐시관리자는 주어진 질의가 일정한 형태를 따르면 그 질의 포함관계를 스플릿 알고리즘과 질의종속그래프를 이용하여 처리하며(라인2-4), 만일 질의가 일정 형태를 따르지 않는다면 그 질의 전체 모양을 질의결과와 함께 처리하는(라인5-25), 2단계 구조로 되어있다. 캐시공간이 부족하게 되면 앞절 4.1과 4.2에서 설명한대로 이득함수의 값이 큰 순서대로 공간

표 1 이득함수를 위한 노드 구조

질의id	타입 (type)	정규화 형태 (canonical form)	레퍼런스 리스트 (reference list)	업데이트 리스트 (update list)	수행비용 (cost)	결과크기 (size)	결과에 대한 포인터 (pointer to RS)	베이스 질의 id (id of base query)	링크된 자손 리스트 (id lists of childeren)

```
1: Check whether the query Q is a canonical form.
2: If Q is not a canonical form then
3:      Do exact string matching for subsumption test, and if RS(Q) is found at the cache,
4:      return RS(Q) from the cache. Else Q is executed on raw database.
5: Else if Q is a canonical form then
6:    call split algorithm for Q, i.e., Q => (Q1, Q2)
7:    If Q1 ∈ Vc, then Q2 is slice query.
8:       Search for RS(q') which has a link --> to Q1 and subsumes Q2.
9:       If a subsuming query q' is found in Vc
10:           Execute Q2 on the subsuming RS(q').
11:   Else
12:           Execute Q2 on RS(Q1), add Q2 into Vnc, and Add Q2 -->Q1 into the graph.
13:   End if
14: Else // Q1 has not been materialized
15:    Let Q11 be IA and Q2' be slice query .
16:    Add Q1, Q2 into Vnc, and Add Q1 -->Q2 into the graph.
17:    Search for a RS(q') which has a link --> to Q11 and subsumes Q2'.
18:    If a subsuming query q' is found in Vc
19:       Execute Q2' on the subsuming RS(q').
20:    Else
21:       Execute Q2' on RS(Q11), add Q2' into Vnc.
22:       Add Q2 -->Q2, Q2' -->Q11 into the graph.
23:    End if
24: End if
25: End if
26: If space  is not enough then
27:    For i=1 to K do
28:       RSi = list of RSs with exactly i references in decreasing profit order
29:    End for
30:    RS = list of RSs in order RSk>RSk-1>...>RS1
31:    C=minimal prefix of RS such that ∑RSj ∈ C sj ≤ space of the cache
32:    Keep RSj ⊂ C and kick any RSi ⊂ RS-C out of the cache.
33 :End if
```

그림 7 이단계 캐시관리자 슈도 코드

이 허용하는 한도내에서 질의결과를 캐시에 유지하고 나머지는 캐시로부터 추출되게 된다(라인26-33).

5. 성능 평가

5.1 실험 개요

정규화형태를 따르지 않는 데이타웨어하우스상의 질의에대한 동적 질의결과캐시교환 기법의 성능평가는 Set Query 벤치마크와 TPC-D 벤치마크를 통해 이전 연구에서 보여졌다[4]. 따라서 본 논문에서는 질의가 정규화형태를 따르는 경우에 대해서 100MB 크기의 토이(toy) 데이타웨어하우스를 이용해 그 성능평가를 한다.

성능평가의 주요 관심사는 다음과 같다. 첫째, 제시된 동적 질의결과캐시교환 기법이 기존의 정적 질의결과캐시교환 기법과 비교해 어떠한 성능 차이가 있는지 살펴본다. 둘째, 질의 스플릿과 질의종속그래프에서 질의수행을 위한 캐시된 질의결과 탐색이 얼마나 바람직하게 이루어지는지를 살펴본다. 마지막으로 동적 질의결과캐시교환 기법에서 질의결과의 캐시내 유지비용이 실제 성능에 어느정도 영향을 미치는지를 살펴본다. 본 논문에서는 [4]에서처럼 질의 수행비용을 하드웨어, 운영체계 및 데이타베이스시스템 등에 따른 물리적 차이를 제거하는 한 방법으로서, 질의 수행이 CPU보다는 디스크 IO 에 바운드됨을 가정하여 전체(논리적, 물리적) 디스크 버퍼에 대한 총 액세스 수로 모델링하였다.

5.2 실험 결과

- 동적 질의결과캐시교환 성능

동적 질의결과캐시교환 기법의 정적 질의결과캐시교환 기법에 대한 성능 비교 실험을 위해 HP-UX 에서의 Oracle 8i 데이타베이스를 이용한 TPC-h [12] 벤치마킹이 이용되었다. 생성된 데이타베이스는 TPC-h 제안의 0.1 스케일인데, 정규화형태의 질의 수행을 위해 데이타베이스 스키마와 질의는 그 의미와 내용은 유지하면서 각각 스타-스키마와 정규화형태 질의로 재구성되었다. 전체 정규화형태 질의 개수는 2만개로서 TPC-h 의 질의 생성자(query generator)의 각 질의 템플릿트에 랜덤값을 적용해 생성하였는데, 총 24개의 데이타큐브 질의로 구성된 큐브 래티스에 대해 질의가 수행될 수 있도록 하였다.

성능 분석의 비교 대상은 [3]에서 제시된 정적 질의결과캐시교환 방법과 이를 보완한 방법 두 가지인데, 본 논문에서는 이를 각각 Static-1, Static-2라 부르도록 한다. Static 1은 캐시될 질의결과의 개수가 n으로 주어졌을때, 각 데이타큐브 질의에 대해 제일 큰 *benefit* 값을 가지는 순으로 캐시될 질의결과(즉 데이타큐브)를 미리 순차적으로 정한다. 여기서의 *benefit*은 본 논문 4절의 benefit과는 그 기본 개념은 비슷하지만 구체적인 방법은 다른 [3]의 함수임을 유의한다. 그들이 제시하는 x의 *benefit*이란 x의 자손 y를 x를 이용해 수행하는 비용($c_{y,on\ x}$)과, 한개씩 캐시될 질의결과를 정하는 1, 2, 3, ..., n 단계별에서 각 단계별로 y를 수행하는데 제일 비용이 적게드는 것으로 알려진 다른 질의결과(z)를 이용해 y를 수행하는 비용(c_{y,on_z})과의 차를 의미한다. 다시 말해 $benefit(x) = c_{y,on\ x} - c_{y,on_z}$이다. 대부분의 현실에서는 Static-1에서처럼 미리 캐시할 질의결과 개수를 정하기보다는 캐시 공간이 정해진 경우가 대부분일 것이므로 성능평가의 공정성을 기하기 위하여, 캐시 공간이 남아있는 한 benefit 순으로 최대한 많은 숫자의 질의결과를 캐시하도록 Static-1을 변형하여 Static-2를 만들었다. 하지만 두 경우 모두 캐시될 질의결과의 집합은 미리 결정해지고, 시간이 지나더라도 변하지않는 정적 캐싱을 하는 것은 동일하다.

성능 비교의 척도는 4.2절에서 제시된 식 (6)의 *비용절감비율*(CSR)이며, 캐시된 질의결과의 유지비용은 본 절의 세번째 실험에서 살펴보므로 고려치 않았다. 이 실험에서는 각 질의는 랜덤하게 형성되더라도 이용되어지는 베이스질의가 유니폼 분포(*uniform distribution*)를 따르는 경우와(그림 8), 특정 베이스질의가 상대적으로 많이 이용되도록 전체 a% 질의는 전체 베이스질의중 b%를 이용하도록 한 Zipf 분포의 경우를 나누어 살피었다(그림 9). Zipf 분포의 예로서 a, b 값으로 각각 70

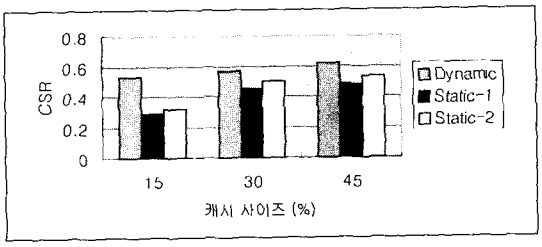

그림 8 유니폼분포를 따르는 경우

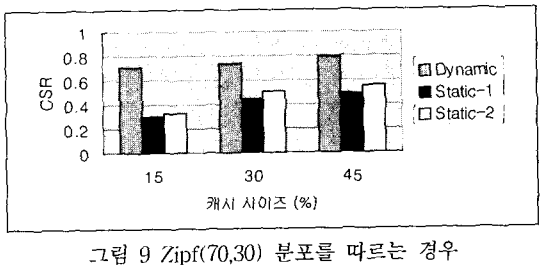

그림 9 Zipf(70,30) 분포를 따르는 경우

과 30을 사용했다. 캐시 사이즈로는 Static-1이 최소한으로 요구하는 크기가 전체 데이타베이스 크기의 15% 이어서 공정한 성능 비교를 위해 최소 15%, 그리고 30%, 45% 로 설정했다.

그림 8,9 모두에서 나타나듯이, 본 논문의 동적 질의결과캐시교환 기법(그림에서 Dynamic으로 표시)은 Static-1, Static-2 모두에 비해 모든 캐시 사이즈에 대해서 비용절감비율이 높게 나타났다. 질의 분포가 유니폼인 경우에는 동적 질의결과캐시교환은 Static-1에 비해서 평균 54%, Static-2에 비해서는 37%가량 나은 비용절감비율을 보인다. 질의 분포가 치우친(skewed) 경우에 동적 질의결과캐시교환은 Static-1에 비해서 평균 94%, Static-2에 비해서는 70%가량 나은 비용절감비율을 보여, 동적 질의결과캐시교환이 질의의 분포가 치우치는 경우에 정적 캐싱에 비해 훨씬 좋은 성능을 보임을 알 수 있다.

- 오버헤드 비용 분석

정규화형태 질의 q_i는 질의스플릿 과정을 통해 캐시된 다른 질의결과 RS_j를 사용하여 수행되어진다. 질의스플릿과 질의종속그래프에서 인터벌트리를 이용한 질의포함관계 검색은 4.2 절에서 살펴보았듯이 폴리노미알(polinomial) 시간복잡도를 가진 CPU 바운드 작업인데, 의사결정질의 수행이 디스크 액세스를 많이 필요로 하는 IO 바운드 작업임을 감안해 스플릿 자체와 질의포함관계 비용은 상대적으로 간과할 수 있다. 더 중요한 사항은 과연 스플릿 과정에 의해 선택된 RS_j가 실제적으로 질의 q_i를 수행하는데 최소한의 수행비용을 제공하는 것인가 하는데 있다.

즉 질의 q_i를 수행시킨 RS_j가 다른 모든 $RS_k(k \neq j)$에 대해서 $c_{ij} \leq c_{ik}$ 인지와, 실제로 최소값을 제공하는 RS_{opt}에 비해서 얼마나 그 수행비용에 있어 차이가 나는지를 검토해 보는 것은 의의가 있는 실험이다. 참고로 실제 환경에서 RS_{opt}를 구한다는 것은 또 다른 최적화 문제이겠지만, K의 크기(질의가 사용할 수 있는 RS_j의 총 개수)가 크지 않은 경우 실험목적상 q_i를 모든 RS_k에 대해 실제 수행하고 그 결과를 비교함으로서 RS_{opt}를 구할 수 있음을 유의하기 바란다. 이 실험에서는 2만개의 랜덤하게 생성된 정규화형태의 질의에 대해 질의스플릿 과정을 통한 RS_j 선택 후 수행비용의 총합(C_{split})과, 같은 셋의 질의들을 각각의 실제 RS_{opt}를 이용하여 수행한 비용의 총합(C_{opt})에 대한 상대값 $\frac{C_{split} - C_{opt}}{C_{opt}}$ 으로 살펴보았고, 캐시 교환에 의한 결과값의 변이를 방지하도록 캐시교환 기법은 수행하지 않았다. 실험은 K를 1부터 23까지 변화하면서 질의스플릿의 성능 스케일링을 살펴보았다.

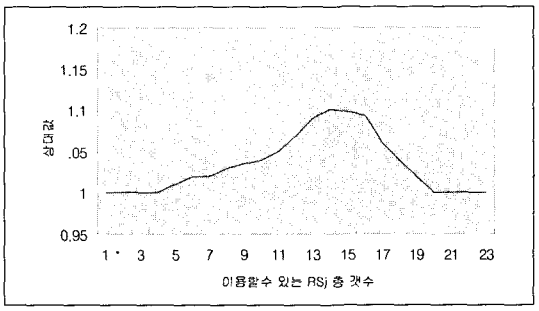

그림 10 C_{opt}와 C_{split}의 상대값 분석

그림 10은 그 결과이다. 상대값이 1.0에 가까울수록 C_{split}이 C_{opt}가 근접함을 의미하는데, 실험 결과는 많은 경우에 있어 상대값은 1.0에 같거나 가까웠고, 최대 차이는 10% 정도임을 보여준다.

- 캐시 유지비용 요소 분석

질의가 기반하는 데이타베이스의 테이블이 업데이트되면 캐시된 해당 질의결과 역시 이를 반영해야 한다. 본 논문에서 제시하는 질의결과캐시교환 기법이 다른 기법과 비교하여 베이스 테이블의 업데이트에 따른 캐시 유지비용이 상대적으로 어떠하고, 또 이득함수에서 사용하는 업데이트비율과 유지비용이 실제로 알고리즘의 성능에 어떠한 영향을 미치는지 살펴보는 것은 의미있는 일이다. 참고로 베이스 테이블의 업데이트가 질의결과에 영향을 미치는지 아닌지를 분석하는 과정도 필요하나 이 문제는 본 논문의 영역 밖이며, 여기에서는 베이스 테이블의 업데이트가 질의결과의 재구성을 야기하고 그 재구성 비용은 점진적재구성(incremental rebuild)이 아닌 질의재수행(re evaluation) 비용을 가정하는 간단한 모델을 사용한다[14].

먼저 첫번째 실험은 TPC h 기반 스키마에서 하나의 테이블(part 테이블)이 업데이트비율(update ratio) x를 가진다하고 본 논문의 동적 질의결과캐시교환 기법의 비용절감비율을 정적 기법과 비교하였다. 여기서 업데이트비율 x란 매 $1/x$ 번째 질의가 수행된 후 테이블이 업데이트 된다는 것으로 정의한다. 예를 들어 업데이트비율 0.01은 총 100 (1/0.01)개의 질의가 수행된 후 테이블이 업데이트됨을 의미한다. 사용된 캐시의 크기는 데이타베이스의 15% 크기를 사용하였다.

그림 11은 서로 다른 업데이트비율에 대해 동적 질의결과캐시교환이 정적 방식(Static 1, Static 2)보다는 비용절감비율 면에서 낫다는 것을 보여준다. 참고로 그림

9와 10의 성능평가 실험에서는 업데이트는 고려치 않은 것이었음을 유의한다. 그림 11보다 흥미로운 실험은 그림 12에 보이는데, 이 실험에서는 동적 질의결과캐시교환에서 사용하는 이득(profit) 함수 $\frac{\sum_{i\in I, k\in K}(\lambda_i \cdot benefit_{ijk}) - \mu_j \cdot c_j^*}{s_j}$ 가 과연 업데이트비용을 위한 요소(term), 즉 $\mu_j \cdot c_j^*$를 고려함으로서 얻는 비용절감비율이 어느 정도인지를 보기 위한 실험이다. 그림 12의 결과는 본 논문에서처럼 이득함수가 업데이트비용을 고려함이 타당함을 보여주며, 만일 캐시유지에 점진적재구성과 같은 비용이 덜 드는 방식을 도입한다면 그 비용절감비율은 더 높아질 수 있음을 짐작하게 하여준다.

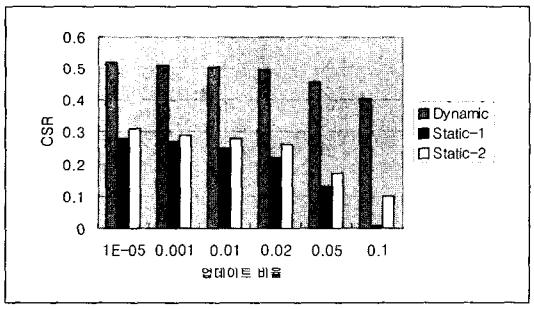

그림 11 업데이트비율의 변환에 따른 CSR 변화

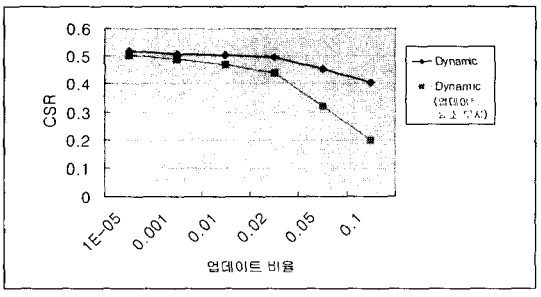

그림 12 이득(profit) 함수에의 업데이트 유지 비용 요소 영향

6. 결 론

본 논문에서는 데이타웨어하우스 환경에서의 의사결정질의의 응답시간을 좋게 하기 위한 하나의 방법으로서 질의결과 캐싱기법을 도입하였다. 의사결정질의중 상당수는 정규화형태를 띄고 있음을 이용하여, 질의결과 캐싱에서 필수적인 효과적 질의포함관계를 위한 질의스플릿과 질의종속그래프를 제시하였고, 또 그 자료 구조와 인터벌트리등을 이용한 질의포함관계 체크 방법등의 구체적 방안이 제시되었다.

질의결과캐싱교환 기법이 최적화해야할 목적함수를 제시하고, 이를 위하여 정적인 방식 대신 동적인 방식의 그리디 휴리스틱을 제안하였다. 이를 위해 먼저 질의들이 정규화형태를 띄지 않은 경우 단순매칭 결과캐싱을 위한 기본 모델을 제시하고, 기본 모델이 어떻게 질의스플릿과 질의종속그래프등을 이용하는 모델로 확장될 수 있는지 보였다.

제시된 기법은 실험을 통해 성능 검증과정을 거치었다. 기존의 정적 질의결과캐싱 기법에 대한 논문에서 제시된 동적 질의결과캐싱 기법의 성능평가뿐 아니라, 질의스플릿과 질의종속그래프를 통한 질의결과포함관계 탐색 기법의 적합성, 질의결과의 캐시 유지비용이 제시된 캐싱기법의 성능에 어느 정도 영향을 미치는지를 각각 살펴보았다.

향후 연구로서는 질의스플릿 이외의 기존에 알려진 다른 질의 변환 및 포함관계를 본 논문에서 제시한 동적 질의결과캐싱 기법에 적용해보는 것과, 단순 전체재구성이 아닌 점진적 재구성방식 등의 적용을 통한 캐시 유지 기법의 일반화, 그리고 질의 수행비용 모델에서 디스크 IO에 한정하지 않은 인덱싱과 버퍼링 등의 다양한 요소를 감안한 모델의 확장 등을 들 수 있다.

참 고 문 헌

[1] W. Inmon, *Building the Data Warehouse, 3rd edition*, John Wiley and Sons, 2002.

[2] J. Han, and M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann, 2001.

[3] V. Harinarayan, A. Rajaraman, and J. Ullman, Implementing Data Cubes Efficiently, *Proc. of the ACM SIGMOD International Conference on Management of Data*, ACM Press, 1996.

[4] P. Scheuermann, J. Shim, and R. Vingralek, WATCHMAN: A Data Warehouse Intelligent Cache Manager, *Proc. of the International Conference on Very Large Databases*, Morgan Kaufmann, 1996.

[5] J. Shim, P. Scheuermann, and R. Vingralek, Dynamic Caching of Query Results for Decision Support Systems, *Proc. of the 11th International Conference on Scientific and Statistical Database Management*, IEEE Computer Society, 1999.

[6] E. O'Neil, P. O'Neil, and G. Weikum, The LRU K Page Replacement Algorithm For Database Disk

Buffering, *Proc. of the ACM SIGMOD International Conference on Management of Data*, ACM Press, 1993.

[7] W.P. Yan and P.A. Larson, Eager Aggregation and Lazy Aggregation, *Proc. of the International Conference on Very Large Databases*, Morgan Kaufmann, 1995.

[8] S. Chaudhuri, R. Krshnamurthy, S. Potamianos, and K. Shim, Optimizing Queries with Materialized Views, *Proc. of International Conference on Data Engineering*, IEEE Computer Society, 1995.

[9] A. Gupta, V. Hariarayan, and D. Quass, Aggregate query Processing in Data Warehousing Environment, *Proc. of the International Conference on Very Large Databases*, Morgan Kaufmann, 1995.

[10] P. Deshpande, and J.F. Naughton, Aggregate Aware Caching for Multi dimensional Queries, *Proc. of the 7th International Conference on Extending Database Technology*, Springer, 2000.

[11] J. Yang, K. Karlapalem, and Q. Li, Algorithms for materialized view design in data warehousing environment, *Proc. of the International Conference on Very Large Databases*, Morgan Kaufmann, 1997.

[12] Transaction Processing Performance Council, *TPC Benchmark H/R*, http://www.tpc.org.

[13] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.

[14] I.S. Mumick, D. Quass, and B.S. Mumick, Maintenance of data cubes and summary tables in a warehouse, *Proc. of the ACM SIGMOD International Conference on Management of Data*, ACM Press, 1997.

심 준 호

서울대학교 계산통계학과(학부), 서울대학교 대학원 전산과학(석사), Northwestern Univ.에서 전산과학(공학박사)을 전공. 경력으로는 미국 Computer Associates Int'l에서 연구개발원, Drexel Univ.에서 Assistant Prof.로서 근무하였으며, 2001년 3월 이후 현재까지 숙명여자대학교 정보과학부 컴퓨터과학과에 조교수로 재직 중. 주요 연구 활동 분야는 데이타베이스, 데이타웨어하우스, 웹, 전자상거래 등