부분 범위 처리를 통한 관계형 데이터베이스 시스템 성능 향상

박경민[0]

고려대학교 컴퓨터정보통신대학원

km20000@nate.com

Partial Range Scan For Increased Relational Database System Performance

Kyungmin Park[0]

Graduate School of Computer & Information Technology, Korea Univ.

요 약

관계형 데이터베이스의 데이터와 처리요청이 증가할수록 해당 데이터의 처리속도는 떨어지게 마련이다. 처리해야할 범위가 넓어도 빠른 속도로 결과를 처리할 수 있다면 데이터베이스 시스템의 효용성의 크게 증대될 것이다. 만약 조건에 맞는 데이터가 100만 건이 나왔다고 한다면 굳이 모든 것을 액세스를 한 다음에 그 결과를 출력할 필요는 없기 때문이다. 그러므로 사람의 눈으로 확인할 일부분만 결과를 먼저 제공하고 나머지는 다음 데이터를 원할 때 처리해서 제공하는 방식은 실제로 처리할 데이터는 아주 소량이 되므로 조건 범위와 무관하게 처리량을 크게 줄일 수 있는 장점이 있다. 본 논문에서는, 관계형 데이터베이스 환경에서 부분 범위처리를 통한 성능향상의 개념과 그 분석을 통한 관계형 데이터베이스 성능 향상 모델을 제시한다. 이는 설계에서부터 애플리케이션 개발에 이르기까지 많은 부분에 성능향상을 미치게 될 것으로 보인다.

1. 서 론

데이터베이스의 데이터양이 증가 하게 되면서 가장 고민하게 되는 부분은 처리할 데이터의 범위가 넓을 때를 위한 대책을 찾는 것이다. 소량의 데이터를 처리 할 때에는 어떻게 사용하든지 걱정 할 것은 없다. 그러나 넓은 범위의 데이터를 처리하려는 순간 많은 고민을 하지 않을 수 없다. 처리하고자 하는 프로세스에는 전체 범위를 모두 처리하여 필요한 가공을 한 후 그 결과를 얻고자 하는 경우도 있지만 온라인 검색처럼 찾고자 하는 범위 중에서 일부라도 먼저 액세스되어지기를 원하는 프로세스도 많이 존재하기 때문이다. 이런 경우에는 처리해야할 범위가 넓다고 하더라도 사람의 눈으로 확인하고자 하는 것이므로 일단 필요한 부분을 처리해서 제공하더라도 문제가 존재되지 않는다. 부분범위 처리는 주어진 모든 데이터를 처리하지 않고 일부만 처리하여 결과를 추출하므로 사용자가 아무리 넓은 범위의 처리를 요구하더라도 아주 빠른 수행 속도를 보장 받는다.

본 논문에서는 부분범위 처리의 개념과 원리, 적용원칙 및 부분범위 처리로 유도하는 여러 방법을 소개하고 그 사례를 통해서 성능향상의 모델을 제시 하고자 한다. 또한, 관계형 데이테베이스 시스템 중 가장 최초로 상용화 되어 널리 사용되고 있는 오라클의 예를 들어 설명을 돕고자 한다.

2. 관련연구

2.1 부분범위처리의 개념 및 원리

부분범위 처리란, 드라이빙 조건을 만족하는 범위를

차례로 스캔하면서 체크조건을 검증하여 성공을 한 건을 운반단위(Array Size)로 보내진다. 운반단위가 채워지면 수행을 멈추고 결과를 추출하는 원리 이다. 그러므로 범위가 넓다고 하더라도 그 범위 중 일부만 처리되므로 빠른 수행 속도를 보장 할 수 있는 것이다.

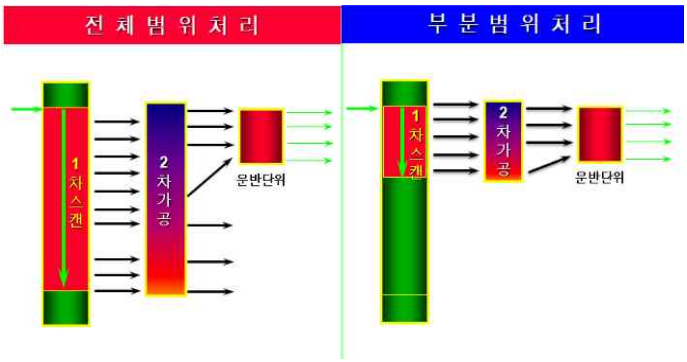

그림 1. 전체범위 처리와 부분범위 처리의 차이

전체범위 처리는 그림 1의 좌측에 나타나 있듯이 드라이빙 조건을 만족하는 범위를 모두 스캔한 후 성공한 건에 대해 임시 저장 공간에 저장한다. 저장이 완료되면 필요한 2차 가공을 한 후 운반단위만큼 추출시키고 다음 요구가 있을 때까지 일단 멈추게 된다.

반면 우측의 부분범위 처리의 경우에는 체크조건을 검증하여 성공한 건을 바로 운반단위로 보낸다. 여기서 나타난 중요한 점은 전체 범위를 하던지, 부분범위를 하던지 언제나 운반단위만 채워지면 일단 멈추게 된다는 점이다. 그러므로 가능하다면 부분범위 처리를 하는 것이 2차가공의 시간을 획기적으로 줄일 수 있는 것이다.

2.2 부분범위처리의 적용범위

부분범위 처리를 적용 할 수 있는 범위는 전체범위를 스캔 후 처리해야하는 경우를 제외한 형태에서 대부분 적용이 가능하다.

```
SELECT SUM(SAL)                          SELECT EMP_NO
FROM EMP                                 FROM EMP1
WHERE EMP_NUM LIKE '200812%';            UNION
                                         SELECT EMP_NO
                                         FROM EMP2;
```

그림 2. 부분범위 처리가 불가능 한 예

위 그림 2의 좌측 SQL은 SELECT List에서 SUM이 존재하고 또한 우측 SQL은 UNION이 존재한다.

만약 SUM이 존재한다면 주어진 조건의 일부만 액세스해서 결과를 얻는 것은 불가능하다. 또한 UNION, MINUS, INTERSECT를 사용한 SQL같은 경우에는 그 결과가 반드시 유일해야 하기 때문에 부분범위로 처리되어질 수 없다. 또한 만일 ORDER BY가 사용되었다면 마찬가지로 전체 범위를 처리 할 수 밖에 없다. 그러나 이는 만일 ORDER BY 를 사용한 컬럼의 해당되는 인덱스가 있다면 이를 통해서 부분범위처리가 가능하기 때문에 해당 인덱스와 SQL 문의 실행계획을 면밀히 살펴보아야한다.

2.3 부분범위처리의 여러 방법

1) 액세스 경로를 이용하여 SORT 대체
2) 인덱스만 액세스하여 처리
3) ROWNUM 을 이용한 부분범위처리
4) FILTER 형 부분범위처리

모든 경우를 부분범위 처리를 할 수는 없지만 위의 방법 등을 통해서 보다 많은 부분범위 처리로의 유도를 가능하게 한다.

2.3.1 액세스 경로를 이용하여 SORT 대체

```
SELECT *                                 SELECT /*+INDEX_DESC (AGENCY_CHK AGENCY_CHK_PK) */ *
FROM AGENCY_CHK                          FROM AGENCY_CHK
WHERE BIZ_NO_NATION LIKE '1234%'         WHERE BIZ_NO_NATION LIKE '1234%';
ORDER BY BIZ_NO_NATION DESC;
```

그림 3. 액세스 경로를 대체하여 SORT한 예

위 그림 3의 SQL을 비교하기에 앞서서 AGENCY_CHK 테이블은 BIZ_NO_NATION 컬럼으로 구성된 Ascending 된 인덱스가 존재한다고 가정한다. 좌측의 SQL은 AGENCY_CHK 의 WHERE절 조건에 만족하는 데이터를 액세스 하여 Decending 하여 운반단위 만큼 처리하고 처리를 마치게 된다. 그러나 우측의 SQL은 최초 Asending되어 있는 인덱스를 역순으로 액세스 하여 WHERE 절에 해당되는 로우들만 운반단위로 보내며 운반단위가 채워지면 추출되게 된다.

```
********************************************************************************

SELECT *
FROM AGENCY_CHK
WHERE BIZ_NO_NATION BETWEEN '123456789' AND '234567890'
ORDER BY BIZ_NO_NATION DESC

Call     Count CPU Time Elapsed Time       Disk      Query    Current        Rows
------- ------ -------- ------------ ---------- ---------- ----------  ----------
Parse        1    0.000        0.000          0          0          0           0
Execute      1    0.000        0.000          0          0          0           0
Fetch       37    0.010        0.007          0        317          0         351
------- ------ -------- ------------ ---------- ---------- ----------  ----------
Total       39    0.010        0.007          0        317          0         351

********************************************************************************

SELECT /*+INDEX_DESC (AGENCY_CHK AGENCY_CHK_PK) */ *
FROM AGENCY_CHK
WHERE BIZ_NO_NATION BETWEEN '123456789' AND '234567890'

Call     Count CPU Time Elapsed Time       Disk      Query    Current        Rows
------- ------ -------- ------------ ---------- ---------- ----------  ----------
Parse        1    0.010        0.000          0          0          0           0
Execute      1    0.000        0.000          0          0          0           0
Fetch       37    0.010        0.002          0        356          0         351
------- ------ -------- ------------ ---------- ---------- ----------  ----------
Total       39    0.020        0.002          0        356          0         351
```

그림 4. 실행결과의 비교

그러므로 좌측의 SQL이 훨씬 향상된 처리속도를 나타냄을 그림4 의 Trace를 통해서 알 수있다.

2.3.2 인덱스만 액세스하여 처리

인덱스는 최초의 로우를 찾을 때만 랜덤 액세스를 실행하게 된다. 그 다음부터는 스캔을 하게 되지만, 테이블을 액세스 할 때는 항상 랜덤 액세스를 하게 된다. 그러므로 수행속도에 많은 부분을 차지하는 테이블 랜덤 액세스를 하지 않고, 인덱스로만 처리를 하게 된다면 훨씬 향상된 처리를 할 수 있다.

```
SELECT /*+ INDEX (CAMPAIGN_DUP_HIST_BK CMPN) */ CMPN_ID, SUM(CMPN_PROC_CD)
FROM CAMPAIGN_DUP_HIST_BK
WHERE CMPN_ID LIKE 'CAM2008%'
GROUP BY CMPN_ID;



SELECT /*+ INDEX (CAMPAIGN_DUP_HIST_BK CMPN2) */ CMPN_ID, SUM(CMPN_PROC_CD)
FROM CAMPAIGN_DUP_HIST_BK
WHERE CMPN_ID LIKE 'CAM2008%'
GROUP BY CMPN_ID;
```

그림 5. 인덱스 액세스 차이 SQL

그림 5 의 SQL은 먼저 힌트를 통해서 인덱스를 다르게 액세스 하도록 하였다. 위쪽 SQL의 CMPN INDEX는 CMPN_ID로 한 개의 컬럼으로 구성이 되어있고, 아래쪽 SQL은 CMPN2의 INDEX를 사용하고 이는 CMPN_ID + CMPN_PROC_CD로 구성이 되어있다. 위 그림 5 의 SQL들을 실행시킨 결과는 왼쪽 그림 6에서 알 수 있다.

SELECT List에서 CMPN_ID, SUM(CMPN_PROC_CD)를 요청하였고 이는 왼쪽 그림6 의 실행계획에 명확히 표기되어진다. 첫 번째 SQL은 CMPN_ID 만 인덱싱된 CMPN을 사용하였고 이는 4298 로우의 테이블을 액세스 하였다. 원래 인덱스는 ROWID와 컬럼 값의 결합으로 구성이 되므로, 첫 번째 SQL의 경우에는 CMPN_ID의 인덱스 스캔을 통해서 알아낸 ROWID로 테이블을 직접 스캔하여 해당 로우를 추출한 것을 알 수 있다.

반면에, 그림 6의 아래쪽 SQL의 실행계획을 보면 테이블 액세스가 없는 부분을 발견 할수 있다. 이는 같은 SQL이지만, CMPN_ID + CMPN_PROC_CD 로 구성되어진 CMPN2의 인덱스를 액세스하였기 때문이다. 즉, SELECT List에서 두 컬럼에 대한 값을 요청을 하였고 인덱스에 해당 두 컬럼의 값이 모두 존재하기 때문에 테이블에 대한 액세스 없이도 같은 결과를 산출하게 된다. 실행 시간을 비교해보면 CMPN2만 액세스를 한 SQL이 74%정도 향상된 것을 알 수 있다. 이와 같은 부분 범위 처리를 위해서는 자주 사용하는 SQL에 가장 알맞은 인덱스의 생성이 중요하다.

```
********************************************************************************

SELECT /*+ INDEX (CAMPAIGN_DUP_HIST_BK CMPN) */ CMPN_ID, SUM(CMPN_PROC_CD)
FROM CAMPAIGN_DUP_HIST_BK
WHERE CMPN_ID LIKE 'CAM2008%'
GROUP BY CMPN_ID

Call     Count CPU Time Elapsed Time       Disk      Query    Current        Rows
------- ------ -------- ------------ ---------- ---------- ----------  ----------
Parse        1    0.000        0.002          0          0          0           0
Execute      1    0.000        0.000          0          0          0           0
Fetch        7    0.020        0.029         76        165          0          55
------- ------ -------- ------------ ---------- ---------- ----------  ----------
Total        9    0.020        0.031         76        165          0          55

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user: USRDBA (ID=83)

Rows     Row Source Operation
-------  ---------------------------------------------------
      0  STATEMENT
     55   SORT GROUP BY NOSORT
   4298    TABLE ACCESS BY INDEX ROWID CAMPAIGN_DUP_HIST_BK
   4298     INDEX RANGE SCAN CMPN (Object ID 87474)


********************************************************************************

SELECT /*+ INDEX (CAMPAIGN_DUP_HIST_BK CMPN2) */ CMPN_ID, SUM(CMPN_PROC_CD)
FROM CAMPAIGN_DUP_HIST_BK
WHERE CMPN_ID LIKE 'CAM2008%'
GROUP BY CMPN_ID

Call     Count CPU Time Elapsed Time       Disk      Query    Current        Rows
------- ------ -------- ------------ ---------- ---------- ----------  ----------
Parse        1    0.010        0.006          0          0          0           0
Execute      1    0.000        0.000          0          0          0           0
Fetch        7    0.020        0.018         25         31          0          55
------- ------ -------- ------------ ---------- ---------- ----------  ----------
Total        9    0.030        0.023         25         31          0          55

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user: USRDBA (ID=83)

Rows     Row Source Operation
-------  ---------------------------------------------------
      0  STATEMENT
     55   SORT GROUP BY NOSORT
   4298    INDEX RANGE SCAN CMPN2 (Object ID 87475)
```

그림 6. 그림5 SQL의 실행계획 및 분석

2.3.3 ROWNUM을 이용한 부분범위처리

ROWNUM은 물리적으로 저장되어진 컬럼이 아니라 모든 SQL에 삽입해서 사용할 수 있는 가상의 컬럼이다. 이 값은 SQL이 실행되는 과정에서 발생하는 일련번호이므로 SQL 수행 시 같은 로우라 하더라도 서로 다른 ROWNUM을 가질 수 있다.

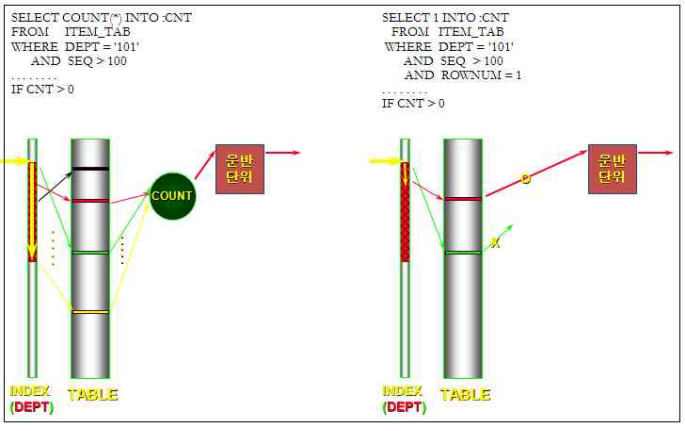

그림 7. ROWNUM 부분처리의 원리

그림 7을 보면 좌측 SQL과 우측 SQL의 실행 결과는 모두 같다. 그렇지만 좌측 SQL과 우측 SQL을 비교하면 WHERE절에 ROWNUM =1 이 다른 것을 발견할 수 있다. 좌측의 SQL은 모든 로우를 스캔한 뒤 그 값으로 IF 문으로 비교하여 사용한다. 이는 IF문에서 한 번에 카운트만 이루어져도 비교가 가능하지만, 필요 없는 스캔이 이루어지는 것이다. 그러므로 우측의 SQL처럼 ROWNUM =1 을 사용한다면 최초의 만족하는 값을 발견하면 SQL이 종료되어 IF문 비교가 이루어진다. 그러므로 최초에 조건에 만족하는 로우가 액세스 될 때까지의 부분만 액세스 되므로 이를 이용한 부분범위 처리가 가능하다. 이 SQL의 경우에는 테이블을 부분범위 처리로 액세스하여 ROWNUM을 COUNT 하다가 주어진 ROWNUM 조건에 도달하면 멈춤(STOPKEY)을 하게 된다.

2.3.3 FILTER 형 부분범위 처리

FILTER 형 부분범위 처리의 경우는 ROWNUM 을 이용한 부분범위 처리와 유사하다. 이 역시 존재의 여부만 판단하는 작업일 경우에 매우 높은 성능을 날수 있다. 즉, 조건을 만족하는 첫 번째를 만나는 순간 실행을 종료하거나 수백, 수천 개의 결과를 가지거나 다를 바가 없다.

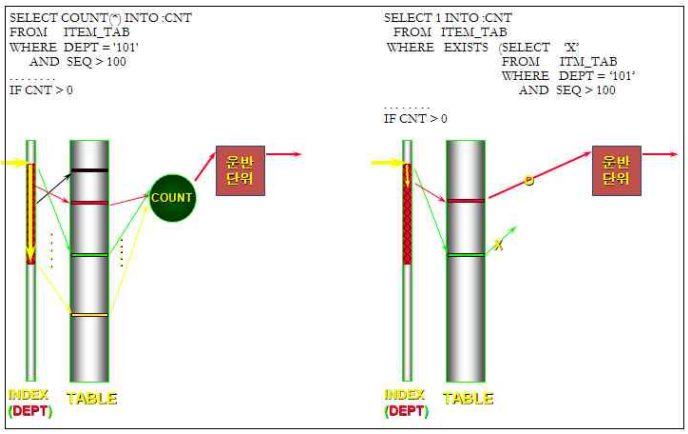

그림 8. FILTER 형 부분범위 처리

그림 8의 EXISTS를 사용하면 EXISTS는 수행결과의 존재여부를 체크하여 성공과 실패만 확인하는 Boolean 함수이기 때문에 단 한번만 성공하더라도 수행을 멈추고 결과를 리턴하게 된다. 이는 그림 8의 좌측에 SQL의 경우에 모든 부분의 스캔의 경우보다 확연히 그 차이를 볼수 있다. FILTER의 경우에는 내부적인 실행 방법이 수행을 하다가 조건을 만족하면 해당 메인 SQL 대상의 로우에 대하여 서브 SQL수행을 멈추게 된다. 그러므로 DUAL 테이블이 단 하나의 로우만 가지게 되므로 실제로는 메인 SQL과 서브 SQL이 단 한번 씩만 수행하게 된다.

3. 결론

관계형 데이터베이스 시스템을 운영하는데 있어서 질의의 속도를 저하 시키는 요인에는 여러 가지 있다. 이러한 요인은 많은 부하와 시스템 운영에 비용의 대부분을 하드웨어 증설 등의 비용으로 충당하게 만든다. 또한, 대부분의 관계형 데이터베이스 시스템에서 가장 많은 속도 저하를 일으키는 요인은 어플리케이션의 SQL 문이 차지하고 있다. 이를 해결하기 위해서는 테이블 저장구조 정확한 전략, 인덱스 활용방안, 액세스 최적방안 수립 등 여러 가지 방법론이 존재한다. 본 논문에서는 액세스 최적방안의 일환으로 부분범위 처리를 통한 성능 향상을 제시 하였다. 특히 본 논문에 예시로 등장하는 SQL은 간단하겠지만 길고 난해해 보이는 SQL도 결국에는 작은 SQL의 모임일 것이고, 이들의 성능을 분석하면서 각각 SQL의 성능을 향상 시키면 결국 관계형 데이터베이스 시스템의 전체적인 안정과 성능향상을 가져 올 것이다.

4. 참고문헌

[1] Database Tuning, Dennis E. Shasha, Prentice Hall , 1992.
[2] Perpect 오라클 실전튜닝, 권순용, 2005.
[3] ORACLE PERFORMANCE TUNING AND OPTIMIZATION , Edward Whalen, 1996.
[4] Oracle9i performance tuning tips & techniques , Niemiec, 2003.
[5] Tuning을 통한 데이터베이스 시스템 성능 향상, 최용락, 1996.