

Oracle Technical Note

Materialized View

Troubleshooting 시리즈는 필자가 한국오라클 고객지원실에서 근무하면서 실제 고객들의 문의 사항이 많았던 부분들에 대해 단편적인 해결책이 아닌 보다 근본적으로 심도 있게 정리한 것이다. 각 호마다 하나의 항목을 정해 기본적인 개념과 메커니즘을 설명한 후 업무 과정에서 발생 빈도가 높은 에러를 처리해 나가는 과정을 설명했다.

지금까지 오라클 메모리 부분, 분산 데이터베이스, 파티션 테이블, 백업/ 리커버리 등에 대해 다루었고, 이번 호에서는 Oracle8의 새로운 개념인 Materialized View(MV)에 대해 살펴보기로 한다. 이 개념은 아직 사용하는 고객은 많지 않으나, 기존의 Cost-based 옵티마이저의 성능을 한 단계 끌어올린 개념으로, 이 개념을 잘 이용하면 SQL문의 수행 속도를 향상시킬 수 있다. 이는 옵티마이저의 주요 기능 중 하나인 Rewrite 기능을 이용하여, 원하는 Summary View를 이용한 Query가 가능하기 때문이다.

글 / 박경희 | 한국오라클 고객지원실 COE팀 |
kyeonghee.park@oracle.com |

Materialized View

오라클 Materialized View는 처음에 대용량의 데이터 웨어하우징에서 Summary 테이블을 Query하는 SQL문의 성능을 증진시키기 위하여 도입된 개념이다. 여기에서 Summary 테이블이란, 우리가 SQL Query 문에서 SUM, MAX, MIN ... 등의 값을 구하는 경우 이들 값을 미리 구하여 다른 테이블에 이들 값을 저장해 놓을 수 있는데, 이때 이 테이블을 일컫는 것이다. 대부분 이들 값을 구하기 위해서는 전체 테이블을 Full Search 하여, Aggregate Operation을 통하여야 하므로 많은 비용이 드는 함수이다. 이들 값을 수행시간에 매번 구한다면 테이블 사이즈가 큰 경우 값을 얻는 시간과 비용이 많이 드는 것은 물론이고, 시스템 전체에 부하를 주게 되어 다른 트랜잭션들에게까지 영향을 미치게 된다. 이런 경우, Materialized View를 이용하면, 대용량 데이터베이스에서 데이터의 Inner/Outer Join을 통한 데이터 수집이 용이하다.

이 Materialized View는 Cost-Based 옵티마이저의 Query-Rewrite와 양방향의 Replication(Snapshot)을 통한 원격 테이블간의 Sub-query를 사용하는 것을 기본으로 하고 있다.

기본 개념

Oracle8i에서 대용량 데이터베이스에서의 Summary 작업의 수행 속도 증대를 위하여 Materialized View를 도입하였다고 언급하였는데, 이는 쉽게 설명하여 Query 실행시간의 수행 속도를 위하여 여러 가지의 Aggregate View를 두어, 미리 비용이 많이 드는 조인이나 Aggregate Operation을 처리하여야 하는 SQL을 위해, 이들을 데이터베이스의 한 테이블로 저장하도록 하는 것이다. 즉, 사용자가 각 지역의 각 제품에 대한 가격의 합을 자주 필요로 하는 경우, 미리 이의 관련 데이터를 한 테이블로 저장하여 필요시 바로 결과를 얻을 수 있도록 하는 것이다. 또한 이는 Cost-Based 옵티마이저의 Rewrite 메커니즘을 적용하여, 사용자가 필요 부분을 Query할 때 미리 만들어져 있는 Aggregate View와 SQL문이 꼭 같지는 않더라도 결론적인 정의만 같으면 자동으로 Summary 테이블을 사용하도록 유도한다.

다음의 SQL문을 보면, 더 쉽게 이해할 수 있을 것이다. 이 Query는 각 Sales에 대한 Order 총계를 구하는 경우이다.

```
SELECT i.ord_ord_id AS order_id,  
       MAX(TO_CHAR(orderdate,'MonthYYYY')) AS orderdate,  
       SUM(i.quantity * p.unitprice) AS total  
FROM orders o, items i, parts p  
WHERE o.ord_id = i.ord_ord_id  
AND p.part_id = i.part_part_id  
GROUP BY i.ord_ord_id;
```

ORDER_ID	ORDERDATE	TOTAL
1	December 2000	7768.25
2	December 2000	3000.2
3	December 2000	4623.5
4	December 2000	1207.5
5	January 2001	1001
...		

만약 이 Query문에 대해 SQL*Plus에서 SET AUTOTRACE ON EXPLAIN 명령어를 적용하여, Explain Plan을 점검해 보면 다음과 같다.

Execution Plan

```
-----  
0   SELECT STATEMENT Optimizer=CHOOSE (Cost=13930  
    Card=4 Bytes=452)  
  1 0  SORT (GROUP BY NOSORT) (Cost=13930 Card=4 Bytes=452)  
    2 1  MERGE JOIN (Cost=13930 Card=333333333 Bytes=37666666629)  
      3 2  SORT (JOIN) (Cost=8552 Card=100000 Bytes=4800000)  
        4 3  MERGE JOIN (CARTESIAN) (Cost=991 Card=100000  
          Bytes=4800000)  
          5 4  TABLE ACCESS (FULL) OF 'PARTS' (Cost=1 Card=10 Bytes=260)  
            6 4  SORT (JOIN) (Cost=990 Card=10000 Bytes=220000)  
              7 6  TABLE ACCESS (FULL) OF 'ORDERS' (Cost=99 Card=10000  
                Bytes=220000)  
                8 2  SORT (JOIN) (Cost=5378 Card=50000 Bytes=3250000)  
                  9 8  TABLE ACCESS (FULL) OF 'ITEMS' (Cost=456 Card=50000  
                    Bytes=3250000)
```

이 경우 Explain Plan을 살펴보면, PARTS 테이블을 Full로 읽은 값과, ORDERS 테이블을 Full로 읽어 이를 Sort한 값을 Merge Join한 후 다시 ITEMS 테이블을 Full로 읽어 Sort한 값을 Merge Join하는 어마어마한 비용이 드는 Query를 하는 것을 알 수 있다 . 그리고 이 각각의 테이블이 큰 경우 매번 이러한 작업을 Sort하고 Merge하는 데 드는 비용을 생각하면 이의 결과를 얻기 위해서는 얼마나 많은 비용과 시간이 드는지를 알 수 있다 .

이 경우, 이들의 SUM과 MAX 값 등의 Summary 정보를 구하는 비용을 줄이기 위해서 대용량 데이터베이스에서는 미리 이러한 Summary 데이터를 계산하여 저장해 둔다. 즉, 수행 시간 때 기초 테이블을 조인하여 Summary 결과를 구하는 것이 아니고, 미리 계산하여 저장해 둬으로써 필요시 보다 빠르게 필요 데이터를 얻을 수 있다.

가령, SALES_SUMMARY 테이블이 각 Orders 테이블에 대해 Order ID, Date, 각 Order에 대한 합을 갖는다면,

```
CREATE TABLE sales_summary AS  
SELECT i.ord_ord_id AS order_id,  
MAX(TO_CHAR(orderdate, 'MonthYYYY' )) AS orderdate,  
SUM(i.quantity * p.unitprice) AS total  
FROM orders o, items i, parts p  
WHERE o.ord_id = i.ord_ord_id  
AND p.part_id = i.part_part_id  
GROUP BY i.ord_ord_id;
```

을 미리 생성해 놓는다. 그리고 Orderdate에 관한 MAX 값과, 각 주문에 대한 비

용이 필요할 때는 기초 테이블인 Orders, Items, Parts 테이블을 직접 Query하는 것이 아니고, 미리 생성해 놓은 Sales_summary 테이블을 이용한다면, 이들 Aggregate Operation을 수행시간에 처리하는 부하가 없어지게 되어 SQL문의 수행 속도를 높일 수 있다.

```
SELECT order_id, orderdate, total
FROM sales_summary;
```

ORDER_ID	ORDERDATE	TOTAL
1	December 2000	7768.25
2	December 2000	3000.2
3	December 2000	4623.5
4	December 2000	1207.5
5	January 2001	1001
...		

그러면, 이의 Execution Plan을 살펴보자.

```
Execution Plan
-----
0  SELECT STATEMENT Optimizer=CHOOSE (Cost=99 Card=10000
   Bytes=340000)
  1 0 TABLE ACCESS (FULL) OF 'SALES_SUMMARY' (Cost=99
     Card=10000 Bytes=340000)
```

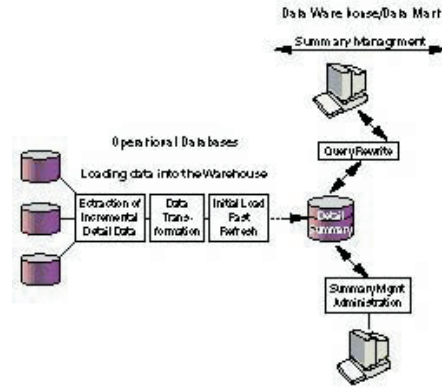
이 Execution Plan을 보면 미리 만들어진 SALES_SUMMARY의 Summary 테이블을 Full로 읽어서 결과를 얻어내기 때문에 훨씬 일하는 양이 줄어들었음을 알 수 있다.

[그림 1] 에서 보듯이 Materialized View를 구성하는 Summary 테이블은 주로 GROUP BY를 포함하는 SQL 문장으로 일반적인 데이터베이스의 인덱스와 특성이 유사하다. 그 이유는, 이를 만들어 두면 QUERY의 수행 속도를 증가시킬 수 있으며, SQL 애플리케이션에서 MV를 사용하는 경우 DBA는 프로그램에 영향을 끼치지 않고 언제든지 생성 및, 제거가 가능하며, Summary 테이블은 공간을 차지하며, 관련 기초 테이블이 변경되는 경우 이 Summary도 변경되기 때문이다.

결론적으로, Summary 오브젝트는 SQL문의 SELECT ... GROUP BY와 같은 SQL문이 1개 또는 그 이상의 테이블과 뷰를 이용하여 SUM, MIN, MAX, AVG, COUNT(*), COUNT(x), COUNT(DISTINCT), VARIANCE 또는 STDDEV의 오퍼레이션을 이용하여 생성하는 것과 같으며, 이 Summary에 인덱스를 생성할 수도 있고, 이를 파티셔닝할 수도 있다.

이 Summary를 Oracle8i에서는 Materialized View라고 부르며, CREATE, ALTER, DROP의 SQL 생성 문장에 의해 생성할 수도 있고 제거할 수도 있다. 그리고, 데이터 웨어하우징 시스템을 가지고 운영하는 많은 업체에서는 이미 자신의 Summary 테이블을 가지고 있을 수 있다. 이 경우는 Materialized View를 다시 생성할 필요 없이 옵티마이저의 Query Rewrite에 의해 기존의 Summary를 계속

사용할 수 있다 .



(그림1) Materialized View를 구성하는 Summary테이블

Materialized View 구현하기

Materialized View를 생성하고 이를 SQL문에서 Query의 수행 속도 개선을 위해 사용하고자 하는 경우 구성 방안은 다음과 같다.

- ① Materialized View를 사용하기 위한 패러미터 설정
- ② Materialized View를 사용하기 위한 권한 부여와 Query Rewrite를 위한 옵티마이저 관련 권한 부여
- ③ Materialized View를 생성하기
- ④ 옵티마이저 통계정보 재설정
- ⑤ Query Rewrite가 제대로 되어 Materialized View를 정확히 사용하였는지의 확인
- ⑥ Materialized View를 Refresh하기

1단계 : Materialized View를 위한 패러미터

Materialized View를 사용하기 위해서는 아래의 패러미터를 initSID.ora 파일에 설정한 후 데이터베이스를 다시 시작하여야 한다. 그리고 아래의 패러미터는 Materialized View값을 Refresh하고, Query Rewrite하기 위해 아래의 권장 값만큼 설정해야 한다.

Materialized View Refresh를 위한 패러미터

- JOB_QUEUE_PROCESSES
Materialized View를 Fresh하기 위해 한 개 이상의 JobQueue (SNP) 프로세스를 사용하기 위함. 1 또는 2를 권장.
- JOB_QUEUE_INTERVAL
Job Queue 프로세스가 Job Queue를 점검하는 시간 간격으로 60초를 권장.

Query Rewrite를 가능하게 해주는 패러미터

- OPTIMIZER_MODE
Materialized View를 사용하기 위해서는 Cost-Based 옵티마이저여야 하므로 ALL_ROWS, CHOOSE, 혹은 FIRST_ROWS 중의 어느 하나를 사용한다.

- `QUERY_REWRITE_ENABLED`
Query Rewrite를 위해 TRUE로 설정한다.
- `QUERY_REWRITE_INTEGRITY`
오라클이 Query Rewrite를 하는 정도에 대해 결정하는 변수이며 이는 기본적으로 ENFORCED이며, Consistency와 Integrity를 보장한다.
- `STALE_TOLERATED`
• 사용되어진 기초 테이블과 Consistent하지 않은 View를 이용한 Query Rewrite를 허용한다.
- `COMPATIBLE`
사용할 수 있는 오라클 함수들의 Compatibility를 결정하는 값으로, 8.1.0 또는 그 이상으로 설정한다.
- `ORACLE_TRACE_COLLECTION_NAME`
Trace를 모으는 파일 이름을 결정하는 것으로, oraclsm이다.
- `ORACLE_TRACE_COLLECTION_PATH`
Trace 파일을 모으는 디렉토리나 폴더로, %ORACLE_HOME%/otrace/admin/cdf이다.
- `ORACLE_TRACE_COLLECTION_SIZE`
Trace Collection 파일의 초기 사이즈이다. 이것은 0 이다.
- `ORACLE_TRACE_ENABLE`
TRUE로 설정시 Trace 파일을 모으는 작업이 가능하다.
- `ORACLE_TRACE_FACILITY_NAME`
Trace Collection Facility 중 Event를 설정하는 것으로, oraclesm 설정시 Summary Event를 설정하는 경우이다.
- `ORACLE_TRACE_FACILITY_PATH`
Trace Facility Definition Files의 위치를 결정하며, 기본적으로 %ORACLE_HOME%/otrace/admin/cdf의 위치에 저장된다.

2단계 : Materialized View와 Query Rewrite를 사용하기 위한 권한 부여
Oracle8i에서는 Materialized View를 생성하고 사용하기 위해 또 Query Rewrite를 위하여 이에 필요한 권한이 주어져야 한다. 기본적으로 Materialized View는 사용되는 기초 테이블을 생성한 User에서 생성할 것을 권장하며, 이 경우 자동으로 이들 기초 테이블을 사용하는 권한이 주어진다. 이외에 Create Materialized View(또는 Create Snapshot), Create Table, Create View, Query Rewrite의 시스템 권한이 필요하다 . 또한 이 User에게 Materialized View를 저장하기 위한 저장 테이블 스페이스를 사용할 수 있는 충분한 Quota가 주어져야 한다 .

3단계 : Materialized View 생성하기

Materialized View를 생성하기 위하여 CREATE MATERIALIZED VIEW의 명령어를 사용한다.

예를 들어, 다음의 SQL문은 ORDERS, ITEMS, PARTS을 기본으로 하는 Materialized View를 생성하는 문장이다.

```
CREATE MATERIALIZED VIEW sales_summary
BUILD IMMEDIATE
REFRESH
COMPLETE
ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT i.ord_ord_id AS order_id ,
MAX(TO_CHAR(orderdate, 'MonthYYYY')) AS orderdate ,
SUM (i.quantity * p.unitprice) AS total
FROM orders o, items i, parts p
WHERE o.ord_id = i.ord_ord_id
AND p.part_id = i.part_part_id
GROUP BY i.ord_ord_id;
```

여기에서 CREATE MATERIALIZED VIEW문에서 사용되는 구문을 살펴보자.

- BUILD IMMEDIATE

이 옵션은 Materialized View를 생성하는 시점에서 이와 관련된 데이터들도 수집하도록 하는 기능이다.

- BUILD DEFERRED

Materialized View를 생성은 하되, 그 안의 데이터는 정상적인 데이터베이스가 영향을 받지 않도록 추후에 수집하도록 하는 기능이다.

- REFRESH

REFRESH 절은 오라클이 Materialized View의 데이터를 언제, 어떻게 Refresh 하는지를 결정토록 하는 방안이다. Refresh 절을 기술할 때는 어떤 타입의 Refresh인지, 어떤 방법으로 Refresh되는지가 기술되어야 한다. Refresh 방법에는 On-commit 방법과 On-demand 방법 2 가지가 있으며, On-commit은 기초 테이블에 Commit이 일어날 때 Refresh가 일어나는 방안이며, 이는 1개의 테이블에 COUNT(*), SUM(*)과 같은 집합 함수가 발생하거나, Materialized View에 조인만이 있는 경우, Group by절에 사용된 컬럼에 대해 COUNT(col)함수가 기술된 경우만 사용이 가능하다.

이에 반해 On-demand는 사용자가 Refresh를 위해 관련 패키지인 DBMS_MVIEW 패키지(REFRESH, REFRESH_ALL_MV, REFRESH_DEPENDENT)를 수행하였을 때 일어난다. 만일 Materialized View가 On-commit에 의해 Refresh되어지면 그 다음의 Refresh들이 AlertSID.log 파일과 Trace 파일에 에러를 남기지 않는지 지속적으로 점검해야 한다. 만일 Commit 시

점에 Refresh가 에러를 유발한 경우(트레이스 파일에서 체크가 가능하다), 사용자는 DBMS_MVIEW 패키지를 사용하여 Materialized View를 다시 수정하여야 한다. 이 작업을 하기 전까지는 Commit 시점에 Materialized View Refresh는 발생하지 않는다. 또한 Refresh를 하는 방법에는 FORCE, COMPLETE, FAST, NEVER의 4가지가 있다.

- COMPLETE: Materialized View의 정의에 따라 Materialized View의 데이터 전체가 Refresh되는 것으로, ATOMIC_REFRESH= TRUE 와 COMPLETE으로 설정한 경우이며, 이는 ATOMIC_REFRESH =FALSE이고 Force로 설정한 경우와 같다.
- FAST: 새로운 데이터가 삽입될 때마다 점진적으로 Refresh되는 방안으로 Direct Path나 Materialized View Log를 이용한다.
- FORCE: 이 경우 먼저 Fast Refresh가 가능한지 점검한 후 가능하면 이를 적용하고, 아니면 Complete Refresh를 적용한다.
- NEVER : Materialized View의 Refresh를 발생시키지 않는다.

- ENABLE QUERY REWRITE

이 옵션이 주어지면만 Oracle8i는 임의의 SQL을 처리할 때 Query Rewrite를 고려한다. 만일 MV 생성시 이를 지정하지 않은 경우는 ALTER MATERIALIZED VIEW를 이용하여 설정하도록 한다.

- AS

일반 View나 Snapshot처럼 AS 구문 뒤에 필요한 컬럼과 조건들을 기술한다.

4단계 : 옵티마이저 관련 통계정보를 Refresh하기

Materialized View에서 사용되어지는 Query-Rewrite 기능은 기존의 Cost-based 옵티마이저를 근간으로 한 것이기 때문에 모든 테이블의 정확한 통계정보를 옵티마이저에게 제공하는 것은 무척 중요하다. 해당 오라클 스키마의 모든 오브젝트에 대한 통계정보를 위하여 ANALYZE 명령어를 사용하거나, DBMS_UTILITY의 ANALYZE_SCHEMA 프로시저, 또는 DBMS_STATS 패키지를 이용한다. 아래의 PL/SQL Block은 SALES_APP 사용자의 모든 테이블과 Materialized View의 통계정보를 구하는 예제이다.

```
BEGIN
dbms_utility.analyze_schema('SALES_APP','ESTIMATE',15);
END;
```

5단계 : Query Rewrite와 Materialized View 사용여부 확인 절차
 이제는 SQL*Plus 세션의 AUTOTRACE ON EXPLAIN을 설정하여, Oracle8i가 해당 SQL을 적절히 Rewrite하여 Materialized View를 사용하였는지 점검한다.

```

SELECT i.ord_ord_id AS order_id,
MAX(TO_CHAR(orderdate,'MonthYYYY')) AS orderdate,
SUM(i.quantity * p.unitprice) AS total

FROM orders o, items i, parts p

WHERE o.ord_id = i.ord_ord_id
AND p.part_id = i.part_part_id
GROUP BY i.ord_ord_id;

```

ORDER_ID	ORDERDATE	TOTAL
1	December 2000	7768.25
2	December 2000	3000.2
3	December 2000	4623.5
4	December 2000	1207.5
5	January 2001	1001
...		

```

Execution Plan
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=99
      Card=10000 Bytes=340000)
1 0    TABLE ACCESS (FULL) OF 'SALES_SUMMARY' (Cost=99
      Card=10000 Bytes=340000)

```

이 Execution Plan은 Oracle8i가 자동적으로 Query를 Rewrite하여 MV를 사용하여 Query하였음을 보여 준다. 더욱이 Oracle8i의 Query Rewrite 기능은 어떤 애플리케이션 변경 없이도 MV를 사용함으로써 Query의 속도 증대에 기여하였음을 보여 준다.

6단계 : Materialized View를 Refresh하기

인위적으로 Materialized View를 Refresh함으로써 기초 테이블에 대해 최근의 집합 데이터를 갖도록 하기 위함이다 . 이는 DBMS_MVIEW 패키지의 REFRESH, REFRESH_ALL_MVIEWS, 또는 REFRESH_DEPENDENT를 Call함으로써 가능하다. 예를 들어, 다음의 PL/SQL 블록은 SALES_APP 사용자의 SALES_SUMMARY Materialized View를 Refresh하는 방법이다 .

```

BEGIN
dbms_mview.refresh('SALES_APP.SALES_SUMMARY');
END;

```

새로운 합계 테이블 생성시

Summary는 CREATE MATERIALIZED VIEW문에 의해 생성되는데, 다음의 예제를 살펴보자. 아래 문장은 khpark_summary라는 뷰인데 이는 제품 이름과 발생 월의 가격에 대한 총합을 구하는 Materialized View이며, khpark이라는 테이블스페이스에 저장된다고 가정하자.

```
CREATE MATERIALIZED VIEW khpark_summary
PCTFREE 0 TABLESPACE khpark
STORAGE(initial 20M next 20M pctincrease 0)
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT t.month, p.prod_name, SUM(f.sales) as sum_sales
FROM time t, product p, fact f
WHERE f.date = t.date AND f.item = p.item
GROUP BY t.month, p.prod_name;
```

위의 SQL로 미루어 khpark_summary MV는 khpark이라는 테이블스페이스에 주어진 STORAGE 옵션으로 생성된다. 즉 초기 EXTENT는 20M의 스페이스를 할당받으며 이 스페이스에 데이터가 모두 저장되는 경우 20M씩 스페이스가 추가 할당되어져 MV를 구성한다. 이때 이 생성 문장이 끝남과 동시에 MV의 데이터가 모두 구성되며, 이는 매번 모든 데이터가 Truncate 후 다시 구성되어지는 방식의 Refresh를 따르며, Query Rewrite가 가능하도록 Enable되어져 있다. 이때 MV에서 구하는 값은 AS 절 이하이다. 그러면 이제 Materialized View 예제의 Explain Plan을 살펴보자.

```
CREATE MATERIALIZED VIEW SALES_SUMMARY
TABLESPACE SALES_TS
PARALLEL (degree 4)      <- Parallel Option 사용해서 생성
BUILD IMMEDIATE
REFRESH FAST
ENABLE QUERY REWRITE
AS
SELECT s.zip_code,p.product_category,sum(s.amount)
FROM SALES S, PRODUCT P
WHERE (s.product_id=p.product_id)
GROUP BY s.zip_code,p.product_category;
```

이 뷰는 Query Rewrite가 가능하게 만들어져 있으며, 이는 옵티마이저로 하여금 Summary 테이블인 Materialized View를 사용 가능하도록 만든다.

이 경우 다음 SQL문에 대해 Explain Plan을 살펴보면, 다음과 같다.

```
SELECT s.zip_code,p.product_category,sum(s.amount)
FROM sales s, product p
WHERE (s.product_id=p.product_id)
GROUP BY s.zip_code, p.product_category;
```

Execution Plan

```
SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=2
1 Bytes=819)
  TABLE ACCESS* (FULL) OF 'SALES_SUMMARY' (Cost=1
  Card=21 Bytes=819)
  PARALLEL_TO_SERIAL SELECT /*+ NO_EXPAND ROWID(A1)
*/A1."ZIP_CODE",
      A1."PRODUCT_CATEGORY",A1."SUM(S.AMOUNT)"
  FROM "TEST"."SALES_SUMMARY" A1 WHERE ROWID
  BETWEEN :B1 AND :B2
```

이 Execution Plan은 Oracle8i가 자동적으로 Query를 Rewrite하여 MV를 사용하여 Query하였음을 보여 준다. 더욱이 Oracle8i의 Query Rewrite 기능은 어떤 애플리케이션 변경 없이도 MV를 사용함으로써 Query의 속도 증대에 기여하였음을 보여 준다.

이 Plan을 살펴보면, 기초 테이블인 SALES, PRODUCT 테이블에 대해 JOIN을 일으켜 SUM 값을 구하는 SQL을 실행하였지만, 이 SQL문은 Query Rewrite를 일으켜 결과적으로 SALES_SUMMARY 테이블을 Parallel Query를 이용하여 Full로 읽어 SUM 값을 구하였다.

기존의 합계 테이블 이용시

이미 데이터 웨어하우징 시스템에서 합계 테이블을 가진 상태이며 이를 계속 Materialized View처럼 이용하고자 하는 경우는 Materialized View를 생성하는 SQL문에 ON PREBUILT TABLE 문을 추가하여 Materialized View를 생성하는데, 이는 새로운 오브젝트를 다시 만드는 것은 아니다.

이 경우 Materialized View의 이름은 기존 테이블 이름과 동일해야 하며, 필요한 SELECT문이 Materialized View 생성시 다시 기술되어야 한다. 예제는 다음과 같다.

```
CREATE MATERIALIZED VIEW sum_of_sales
ON PREBUILT TABLE WITH REDUCED PRECISION
ENABLE QUERY REWRITE
AS
SELECT t.month, p.prod_name, SUM(f.sales) as sum_sales
FROM time t, product p, fact f
WHERE f.date = t.date AND f.item = p.item
GROUP BY t.month, p.prod_name;
```

이 경우 Materialized View는 이 SQL문에 의해 미리 계산되어져 테이블에 저장되며, 이는 원 테이블인 TIME, PRODUCT, FACT 등과는 별도의 Storage Option을 가지고 저장되어진다. 또한 이 Materialized View에 인덱스를 생성할 수도 있다. 또 이 Materialized View는 파티셔닝될 수도 있으며, Materialized View의 데이터는 기초 테이블 변경시 자동으로 또는 수동으로 재계산되어 저장된다.

Materialized View 사용을 위한 Query Rewrite

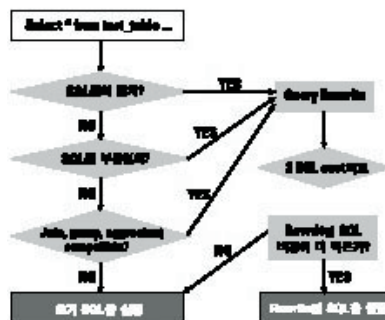
Materialized View를 생성시킨 후에 사용자가 기본 테이블에 대해 합계 또는 조인의 결과를 얻고자 하는 경우 Oracle8i는 Query의 Response Time을 짧게 하기 위하여 Query Rewrite를 통하여 Materialized View를 이용 가능하게 한다. Oracle8i의 Query Rewrite 기능을 위하여 다음의 기능이 필요하다. 인스턴스 패러미터인 OPTIMIZER_MODE, QUERY_REWRITE_ENABLED, QUERY_REWRITE_INTEGRITY, COMPATIBLE을 initSID.ora에 설정한다.

또한, CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW 명령어에 ENABLE QUERY REWRITE 옵션을 추가한다. Materialized view를 생성한 오라클 유저는 반드시 QUERY REWRITE의 시스템 권한이 있어야 한다. 이 권한이 부여되면, 옵티마이저는 모든 SELECT 문에 대해 Query Rewrite를 고려하는데, 모든 SELECT 문뿐만 아니라 CREATE TABLE ... AS SELECT 문, INSERT INTO ... SELECT 문, Set Operator (UNION, UNION ALL, MINUS, and INTERSECT)의 Sub-query도 포함된다. 그림 2는 Oracle8i의 옵티마이저가 어떻게 Query Rewrite를 하는지를 보여 준다.

먼저 옵티마이저는 해당 Query의 조건을 만족하는 Materialized View를 찾아 존재하지 않는 경우 사용자가 처음 실행한 SQL을 실행하고, MV가 있는 경우 Materialized View를 이용한 Query Rewrite를 실행한다.

그 후 처음 SQL과 Rewrite된 SQL과의 Cost 비교를 실행하여 다시 쓰여진 SQL의 비용이 낮은 경우는 새롭게 Rewrite된 SQL문을 실행하고 아닌 경우 원래의 기초 테이블을 이용한 SQL문을 실행한다.

그러므로 이 Materialized View에 Query가 요구하는 조건들이 들어 있어야 하는 것은 필수이다. 위의 그림에서 살펴보았듯이 옵티마이저는 Materialized View에 들어 있는 정보를 최대한 이용하는 여러 가지의 테크닉을 이용하여 Query Rewrite를 실시한다.



(그림 2) Oracle 8i 옵티마이저의 Query Rewrite

완전 일치

가장 간단한 Query Rewrite는 Summary의 정의와 Query의 정의가 정확히 일치하는 경우이다. 이 의미는 WHERE 절의 Join과 Group by절의 컬럼이 정확히 일치하는 경우이다.

아래의 예제를 살펴보자.

```
SELECT t.month, p.prod_name, SUM(f.sales)
FROM time t, product p, fact f
WHERE f.date = t.date AND f.item = p.item
GROUP BY t.month, p.prod_name
HAVING SUM(f.sales) > 10000;
```

이 SQL문은 기존의 Summary 테이블을 사용하기 위하여 아래처럼 Rewrite된다.

```
SELECT s.month, s.prod_name, s.sum_sales
FROM sumtab s
WHERE s.sum_sales > 10000;
```

부분 일치

SQL문 전체가 똑같이 일치하는지의 점검에서 실패하는 경우 옵티마이저는 FROM 절 이하의 나머지 부분이 Materialized View를 정의하는 부분과 일치하는지 점검한다.

다음의 Query는 매월 발생한 총금액을 요구하는 SQL문이다.

```
SELECT t.month, SUM(f.sales)
FROM time t, fact f
WHERE f.date = t.date
GROUP BY t.month
HAVING SUM(f.sales) > 10000;
```

이 경우 Prod_month 라는 Summary Table이 있어서 매월 각 Product에 대해 발생한 총 금액이 있다고 가정하고, FACT 테이블과 PRODUCT 테이블이 1:N 관계인 경우 이 Summary 테이블은 Month로 Group by한 경우 같은 결과를 얻을 수 있어 아래처럼 Rewrite된다.

```
SELECT s.month, SUM(s.sum_sales)
FROM prod_month s
GROUP BY s.month
HAVING SUM(s.sum_sales) > 10000;
```

Summary JoinBack

때때로 SQL Query 문이 Summary 테이블에 있지 않은 컬럼을 참조하는 경우, 오라클은 이 컬럼 값을 다른 Dimension 테이블에서 얻을 수 있는지 않은지 점검 후

Query Rewrite를 실시한다. 예를 들면, 다음과 같다.

```
SELECT t.month, p.prod_name, SUM(f.sales)
FROM time t, product p, fact f
WHERE f.date = t.date AND
f.item = p.item AND
t.quarter = 3
GROUP BY t.month, p.prod_name;
```

이 SQL문은 sumtab에 있지 않은 t.quarter 컬럼을 참조하나, t.quarter는 Hierarchical 구조에 의해 t.month에 의존한다고 한다면 이 컬럼 값은 원래 SQL문의 한 테이블인 Time 테이블과의 Join을 통하여 구해질 수 있도록 Rewrite된다.

```
SELECT s.month, s.prod_name, s.sum_sales
FROM sumtab s
WHERE s.month in (SELECT t.month
                  FROM time t
                  WHERE t.quarter = 3);
```

Summary Rollup

SQL의 Query가 Summary 테이블에 저장된 SUM Value보다 위의 값의 SUM을 요구하는 경우 결과를 얻기 위해 Rolling up을 이용하여 Query Rewrite를 한다.

```
SELECT t.year, p.prod_name, SUM(f.sales)
FROM time t, product p, fact f
WHERE f.date = t.date AND f.item = p.item
GROUP BY t.year, p.prod_name;
```

이는 다음처럼 다시 쓰여진다

```
SELECT v.year, s.prod_name, SUM(s.sum_sales)
FROM sumtab s,
     (SELECT distinct t.month, t.year FROM time t) v
WHERE s.month = v.month
GROUP BY v.year, s.prod_name;
```

이 경우 t.yesr를 얻기 위해 sumtab과 다시 Join을 하고, t.year를 얻기 위해 Rollup을 이용한다.

일반적인 방법

전체 SQL문과 일부 SQL문이 모두 일치하지 않는 것으로 결론이 나는 경우 옵티마이저는 Materialized View를 정의하는 컬럼과 Query의 SELECT 리스트 부분이 일치하지는 않는지, Materialized View와 처음 SQL의 조인 조건이 일치하지는 않는지, Group by절이나 Aggregate 함수 부분이 일치하지는 않는지를 점검하여 Rewrite 여부를 다시 한 번 검토한다. 이 경우 Primary Key, Foreign Key, Not Null Integrity Constraints가 있는 경우 Query Rewrite할 수 있는 기회를 더 높일 수 있다.

올바른 결과가 얻어졌는지의 점검

사용자가 실행한 SQL Query문이 SQL문에서 사용한 기초 테이블을 사용하지 않고, Materialized View를 사용하도록 Rewrite되어 Materialized View를 이용하여 결과를 얻은 경우 잘못된 결과가 도출될 수 있다. 이런 경우는 대체적으로 다음과 같은 경우 발생할 수 있다.

- Materialized View 데이터가 기초 테이블의 데이터와 시점이 동일하지 않은 경우이다. 즉, Summary를 위한 Refresh 작업이 어떤 원인에 의하여 기다리는 상태의 일종인 Pending 상태에 빠졌거나, Complete Refresh를 요청했음에도 Summary 테이블의 몇 로우가 삭제되어 Fast Refresh가 일어난 경우이다.
- Summary 테이블이 더 이상 존재하지 않는 기초 테이블의 데이터를 갖는 경우로, 이는 기초 테이블의 데이터가 삭제되었을 때 Rolling Summary에 발생하는 일반적인 문제이다.
- 테이블의 관계가 부모-자식 관계로 묶여 있는 경우, 조인되는 컬럼이 참조 무결성을 위반한 경우는 어떤 자식 테이블 부분의 로우는 부모 테이블쪽에 Roll Up되지 않는다.
- 어떤 이유로 Summary 테이블에 저장된 값이 중복되거나 부정확한 경우이다.

이 Materialized View가 재계산되는 방법은 Materialized View 생성시 지정할 수 있다.

Materialized View Refresh

이 재계산 방법은 매번 테이블 전체의 데이터가 재계산되는 Full Refresh 방안, DML 문에 의해 기초 테이블의 데이터가 수정될 때마다 변경되는 Fast Refresh 방안, Full Refresh가 되어질 때는 기존의 데이터를 모두 없앤(Truncate) 후 Materialized View의 정의에 따라 모든 데이터를 다시 생성하는 방안이며, Fast Refresh는 Materialized View Log에 의해 변경된 사항만 적용하는 방안이다. 또한 Fast Refresh는 Materialized View를 점진적으로 추가하는 방식을 이용하여 수정한다.

예를 들어, Materialized View가 SUM() 함수에 의해 전체 합을 계산한다면, 새로운 로우가 생성된 경우 이 합은 기존의 SUM() 값에 새로운 값이 추가로 더해지는 방식이다. 이 방식은 기본 테이블에 값이 생성 또는 변경 후 Commit되는 순간 Materialized View의 값이 변경되어진다. 이처럼 변경되는 프로시저는 DBMS_MVIEW 패키지를 이용하여 Refresh된다. 특히 DBMS_MVIEW.REFRESH_ALL_MVIEWS 프로시저는 모든 Materialized View를 Refresh한다.

Refresh를 실행하는 방법에는 ON COMMIT과 ON DEMAND의 2가지가 있으며, 이 실행 방법에 따라 Materialized View의 타입도 결정된다.

- ON COMMIT: COMMIT 시마다 Refresh가 일어나는 것으로 1개 테이블을 Aggregate하고, Materialized View가 조인만 가진 경우에 사용 가능하다.
- ON DEMAND: 사용자가 DBMS_MVIEW 패키지(REFRESH, REFRESH_ALL_MVIEWS, REFRESH_DEPENDENT)를 실행한 경우 Refresh되는 경우이다.

만일 Materialized View가 Commit 시점에 Refresh를 실패하면 사용자는 Refresh 프로시저를 행시켜야 한다. 또한 Refresh 때 어떤 방법으로 Refresh할지를 결정할 수 있다. 여기에는 앞에서 언급하였듯이, FORCE, COMPLETE, FAST, NEVER의 4가지가 있다. Fast Refresh를 이용할 때 주의할 점은 일반적으로 Fast Refresh가 Complete Refresh보다 훨씬 빠르지만, 다음의 경우에는 사용이 불가능하다는 점이다.

- 마지막으로 Complete Refresh가 실행된 이후 Fact 테이블에 DLM이 실행된 경우
- MV가 기초 테이블에 대한 JOIN을 갖는데, JOIN 되는 기초테이블이 View인 경우
- MV가 COUNT(*) 없이 AVG (*)를 갖는 경우
- MV가 COUNT(*)와 SUM (*) 없이 VARIANCE(x)만 갖는 경우

Refresh 과정 중 때때로 Materialized View가 제거 또는 변경(Drop, Alter)된 경우, 권한에 문제가 생긴 경우, Dimension 테이블에 어떤 변화가 있는 경우, MV 테이블은 Invalid 상태에 빠지게 되는데, 이 경우는 ALTER MATERIALIZED VIEW COMPILE 명령어를 통해 Materialized View를 Rebuild해야 한다. 이처럼 Summary 테이블을 유지하는 작업은 한편으로는 DBA에게는 큰 부담일 수 있다. 특히 데이터를 처음에 Loading하는 문제와 이를 지속적으로 Update하기 위한 작업은 커다란 작업으로 때로는 부담으로 다가올 수도 있다. 이를 위해 DBA는 Summary 테이블을 만들고 유지하는데 어느 정도의 시간이 소요되는지 등의 정보를 정확히 파악하고 있어야 한다.

Troubleshooting

ORA-905

오라클 버전 8.1.x 이전에서 Index-Organized Materialized View를 만들 때 발생한다. Oracle8i 이전에는 CREATE MATERIALIZED VIEW는 ORGANIZATION INDEX를 가질 수 없었다. 즉, 이전 버전에서 IOT를 이용한 Materialized View를 생성시키는 ORA-905를 발생시킨다.

- 테이블 생성을 위하여

```
SQL> create table t1 (  
  2 col1 number primary key,  
  3 col2 varchar2(255));  
Table created.
```

- IOT 생성을 위하여

```
SQL> create table t1_iot (  
  2 col1 number,  
  3 col2 varchar2(255),  
  4 constraint t1_iot_pk primary key (col1))  
  5 organization index tablespace userdata  
  6 including col1 overflow tablespace userdata;  
Table created.
```

- IOT MV 생성을 위하여

```
SQL> create materialized view mv_t1  
  2 organization index  
  3 as select * from t1;  
Materialized view created.
```

- IOT MV의 Overflow 지정을 위하여

```
SQL> create materialized view mv_t1_iot  
  2 organization index  
  3 including col1 overflow tablespace userdata  
  4 refresh with primary key  
  5 as select * from t1_iot;  
Materialized view created.
```

- 데이터 디렉터리의 점검을 통한 확인

```
SQL> select table_name, iot_name, iot_type from dba_tables  
  2 where owner = 'TEST'  
  3 and (table_name like '%T1%' or iot_name like '%T1%')  
  4 order by table_name;
```

TABLE_NAME	IOT_NAME	IOT_TYPE
MV_T1		IOT
MV_T1_IOT	IOT	
SYS_IOT_OVER_24894	T1_IOT	IOT_OVERFLOW
SYS_IOT_OVER_24900	MV_T1_IOT	IOT_OVERFLOW
T1		
T1_IOT		IOT

6 rows selected.

- MV의 Summary를 확인하기 위하여

```
SQL> select table_name, master, can_use_log, refresh_method
2 from dba_snapshots
3 where master in ('T1_IOT','T1');
```

TABLE_NAME	MASTER	CAN REFRESH_MET
MV_T1	T1	YES PRIMARY KEY
MV_T1_IOT	T1_IOT	YES PRIMARY KEY

ORA-12051

ON COMMIT attribute is incompatible with other options.

ORA-12054

Cannot set the ON COMMIT refresh attribute for the materialized view.

ORA-12051과 ORA-12054 모두 ON COMMIT Refresh를 수행할 수 없는 상황으로, 구사된 문법의 오류가 있는지를 확인하여야 한다. ON COMMIT Refresh 사용시의 제약 사항은 앞에서 설명하였다.

ORA-928

missing SELECT keyword.

오라클 옵티마이저가 Query Rewrite를 실시할 때 발생하는 것으로, SET Operation이 들어 있는 경우에 발생한다. 이 경우 ALTER {SESSION|SYSTEM} DISABLE QUERY REWRITE로 Query Rewrite를 없애거나, REWRITE(mv) 힌트를 SET Operation마다 부여하는 방안이 있다.

ORA-12016

다음의 실전 TEST를 통해서 에러를 살펴보자. 이 에러는 MV Log 생성시 Primary Key를 만든 경우 MV 생성시 Primary key를 Select 절에 포함시키지 않은 경우이다.

```
SQL> CREATE TABLE promo (
  promotion_key    integer ,
  display_type    varchar2(20),
  promo_cost      number,
  promo_start_date date,
  promo_end_date  date )
  pctfree 0
  pctused 99
  tablespace dim
  storage (initial 8k next 8k pctincrease 0) ;
Statement processed.

SQL> CREATE INDEX promotion_pk_index ON promo (promotion_key)
  pctfree 5
  tablespace indexes
  storage (initial 8k next 8k pctincrease 0) ;
Statement processed.

Primary Key 추가
SQL> ALTER TABLE promo
  ADD CONSTRAINT pk_promotion PRIMARY KEY (promotion_key);
Statement processed.
SQL> drop materialized view log on promo;
Statement processed.
SQL> CREATE MATERIALIZED VIEW LOG on PROMO
  with primary key
  (display_type,promo_cost,promo_start_date,promo_end_date)
  including new values;
Statement processed.
SQL> drop materialized view promo_cst_sum;
```

ORA-12003

Materialized view “GROCERY”.”PROMO_CST_SUM” does not exist.

```
SQL> CREATE MATERIALIZED VIEW promo_cst_sum
  PCTFREE 0 TABLESPACE summ
  STORAGE (initial 2k next 2k pctincrease 0)
  BUILD DEFERRED
  REFRESH FAST
  AS
  SELECT promo_cost, display_type FROM promo;
```

```
FROM promo
*
```

ORA-12016

Materialized View does not include all primary key columns.

```
SVRMGR> drop materialized view log on promo;
Statement processed.
SQL> CREATE MATERIALIZED VIEW LOG on PROMO
      with primary key
      (promotion_key,display_type,promo_cost,promo_start_date,promo_end_date)
      including new values;
CREATE MATERIALIZED VIEW LOG on PROMO
*
```

ORA-12026

Invalid filter column detected.

```
SQL> CREATE MATERIALIZED VIEW LOG on PROMO
      with primary key;
Statement processed.
SQL> drop materialized view promo_cst_sum;
drop materialized view promo_cst_sum
*
```

ORA-12003

Materialized View "GROCERY". "PROMO_CST_SUM" does not exist.

```
SQL> CREATE MATERIALIZED VIEW promo_cst_sum
      PCTFREE 0 TABLESPACE summ
      STORAGE (initial 2k next 2k pctincrease 0)
      BUILD DEFERRED
      REFRESH FAST
      AS
      SELECT promo_cost, display_type
      FROM promo;
FROM promo
*
```

ORA-12016

Materialized View does not include all primary key columns.



한국오라클(주)

서울특별시 강남구 삼성동 144-17
삼화빌딩
대표전화 : 2194-8000
FAX : 2194-8001

한국오라클교육센터

서울특별시 영등포구 여의도동 28-1
전경련회관 5층, 7층
대표전화 : 3779-4242~4
FAX : 3779-4100~1

대전사무소

대전광역시 서구 둔산동 929번지
대전둔산사학연금회관 18층
대표전화 : (042)483-4131~2
FAX : (042)483-4133

대구사무소

대구광역시 동구 신천동 111번지
영남타워빌딩 9층
대표전화 : (053)741-4513~4
FAX : (053)741-4515

부산사무소

부산광역시 동구 초량동 1211~7
정암빌딩 8층
대표전화 : (051)465-9996
FAX : (051)465-9958

울산사무소

울산광역시 남구 달동 1319-15번지
정우빌딩 3층
대표전화 : (052)267-4262
FAX : (052)267-4267

광주사무소

광주광역시 서구 양동 60-37
금호생명빌딩 8층
대표전화 : (062)350-0131
FAX : (062)350-0130

고객에게 완전하고 효과적인
정보관리 솔루션을 제공하기 위하여
오라클사는 전 세계 145개국에서
제품, 기술지원, 교육 및
컨설팅 서비스를
제공하고 있습니다.

<http://www.oracle.com/>
<http://www.oracle.com/kr>