4주차

JOIN

누적매출 구하기

```
select
  T1.지점, T1.판매월, MIN(T1.매출액) 매출액, SUM(t2.매출액) 누적매출액
from 누적매출 t1, 누적매출 t2
where t2.지점 = t1.지점
and t2.판매월 <= t1.판매월
group by t1.지점, t1.판매월
```

선분이력 끊기

```
-- 월말 기준으로 선분을 끊는 경우
select CUSTNO,
TO_CHAR(GREATEST(STARTDT, TO_DATE('20190801', 'YYYYMMDD')), 'YYYYMMDD')
STARTDT,
TO_CHAR(LEAST(ENDDT, TO_DATE('20190801', 'YYYYMMDD')),'YYYYMMDD') ENDDT,
RATIO
from halin
where custno = 'C101'
AND STARTDT <= TO_DATE('20190831', 'YYYYMMDD')
AND ENDDT >= TO_DATE('20190801', 'YYYYMMDD')
```

데이터 복제를 통한 소계 구하기

```
select
a.deptno
,decode(b.no, 1, to_char(empno), 2, '부서계') 사번
,sum(a.sal) 급여합
,round(avg(a.sal)) 급여평균
from emp a, (
  select rownum no
  from dual
  connect by level <= 2
)b;
group by a.deptno, b.no, decode(no, 1, to_char(empno), 2, '부서계')
order by 1, 2
```

ROLLUP 함수 활용 단순하게 작성

```
SELECT DEPTNO
, CASE WHEN GROUPING(EMPNO) = 1 AND GROUPING(DEPTNO) = 1 THEN '총계'
WHEN GROUPING(EMPNO) - 1 THEN '부서계'
ELSE TO_CHAR(EMPNO)
,SUM(SAL) 급여합
,ROUND(AVG(SAL)) 급여평균
FROM EMP
GROUP BY ROOLUP(DEPTNO, EMPNO)
ORDER BY DEPTNO, 사원번호
```

GROUPING SET

```
-- UNION ALL
-- 일별주문내역
SELECT
  SUBSTR(ORDER_DT, 1, 8)
  ,CUST_NO
  ,SUM(ORDER_PRICE)
  FROM T_ORDER74
  WHERE ORDER_DT >= '20160601'
  AND ORDER_DT < '20160701'
  GROUP BY SBSTR(ORDER_DT, 1, 8), CUST_NO
UNION ALL
-- 월별주문내역
SELECT
  SUBSTR(ORDER_DT, 1, 6)
  ,CUST_NO
  ,SUM(ORDER_PRICE)
FROM T_ORDER74
  WHERE ORDER_DT >= '20160601'
  AND ORDER_DT < '20160701'
  GROUP BY SBSTR(ORDER_DT, 1, 6), CUST_NO


-- grouping set 함수

SELECT X.구분
, DECODE(X.ORDER_NM, NULL, X.ORDER_DT, X.ORDER_MM) 일자
, CUST_NO, ORDER_PRICE_SUM
FROM
(
  SELECT DECODE(TO_CHAR(ORDER_DT, 'YYYYMM'), NULL, '일별', '월별') 구분
  , TO_CHAR(ORDER_DT, 'YYYYMMDD') ORDER_DT
  , TO_CHAR(ORDER_DT, 'YYYYMM') ORDER_MM
  , CUST_NO
  , SUM(ORDER_PRICE) ORDER_PRICE_SUM
  FROM T_ORDER74
  WHERE ORDER_DT BETWEEN TO_DATE('20160601', 'YYYYMMDD') AND
```

```
TO_DATE('20160630 235959', 'YYYYMMDD HH24MISS')
  GROUP BY GROUPING SETS(
  (TO_CHAR(ORDER_DT, 'YYYYMMDD'), CUST_NO) -- 일별 grouping,
  (TO_CHAR(ORDER_DT, 'YYYYMM'), CUST_NO) -- 월별 grouping
  )
)X
ORDER BY DECODE(구분, '일별', 1, 2), ORDER_DT, ORDER_NM, CUST_NO
```

최종 출력 건에 대해서만 조인하기

1. TOP N 쿼리

```
-- 일반적인 게시판 출력 조인 방식
select no, 주문번호, 주문일자, 주문금액, 주문고객명, 판매부서명, 판매사원명
from (
  select rownum no, 주문번호, 주문일자, 주문금액, 주문고객명, 판매부서명, 판매사원명
  from (
    select o.주문번호, o.주문일자, o.주문금액
        , c.고객명 주문고객명
        , d.부서명 판매부서명
        , e.사원명 판매사원명
    from 주문 o, 고객 c, 사원 e,  부서 d
    where o.판매부서번호 = 'D004'
      and c.고객번호 = o.주문고객번호
      and e.사원번호 = o.판매사원번호
      and d.부서번호 = o 판매부서번호
    order by o.주문일자, o.판매사원번호
)
  where rownum <= 41
)
where no between 31 and 40
order by no;
```

2. 아래의 두 SQL의 일량은 동일하다

```
-- 조인을 메인쿼리에서 하는 것으로 조인부하를 줄이기
-- ACCESS TABLE BY USER ROWID
select 주문번호, 주문고객번호, 판매사원번호, 판매부서번호, 주문금액
from (
  select rowid r_id
  from 주문 n
  Where 판매부서번호 = 'D004'
)X , 주문 O
where O.rowid = x.r_id;

-- ACCESS TABLE BY INDEX ROWID
select 주문번호, 주문고객번호, 판매사원번호, 판매부서번호, 주문금액
```

```
from 주문
where 판매부서번호 = 'D004'
```

3. 인덱스 구성에 따라 소트에 의한 TABLE RANDOM ACCESS 수가 달라진다.

```
-- IDX : 판매부서번호 + 주문일자
from
(
select r_id, no
from (
  -- 정렬 조건 컬럼인 판매사원번호가 인덱스에 존재하지 않는다.
  -- 소트연산과 테이블랜덤엑세스가 발생한다.
  select rowid r_id, rownum no
  from 주문 n
  Where 판매부서번호 = 'D004'
  order by 판매사원번호, 주문일자
)
WHERE ROWNUM <= 41
)X , 주문 O, 고객 C, 사원 E,부서 D
WHERE X.NO BETWEEEN 31 AND 40
AND O.ROWID = X.R_ID
and c.고객번호 = o.주문고객번호
and e.사원번호 = o.판매사원번호
and d.부서번호 = o 판매부서번호
ORDERBY NO;
```

인덱스 최후행 컬럼에 판매사원번호를 추가하면 (IDX : 판매부서번호 + 주문일자 + 판매사원번호) 정렬을 위한 테이블랜덤엑세스가 발생하지 않는다.

점이력

변경이력만 있다.

```
-- 부서별 최종입사자
SELECT *
FROM(
  SELECT EMPNO, JOB, DEPTNO, TO_CHAR(HIREDATE, 'YYYY.MM.DD') HIREDATE,
  ROW_NUMBER() OVER (PARTITION BY DEPTNO ORDER BY HIREDATE DESC) ROWNUMBER
  FROM EMP
)
WHERE ROWNUMBER = 1;
```

아크(상호배타적 관계의 조인)

외래키 분리 방법

OUTER 조인을 통해 OUTER 테이블의 컬럼값이 NULL 인 경우 조인하지 않도록 함

외래키 통합 방법

통합된 컬럼 (예 : 상품권 번호) 는 항상 값이 존재하기 때문에 상호배타적인 두 테이블에 모두 조인을 시도한다.

각각의 배타적 테이블에 대해 구분코드 (예: 상품권 코드)를 조건으로 각각 조인하고 union all 한다.

인덱스가 [일자 + 구분코드] 일 경우 Decode 함수를 활용해 특정 구분코드에 맞는 테이블과 OUTER 조인하도록 한다.

인덱스가 [구분코드 + 일자] 일 경우 구분코드에 따라 별도로 조인한 후 union all 한다.

징검다리 테이블

중간 조인 테이블에 조인 범위를 줄일 수 있는 필터가 많지 않은 경우.

select 절에 조회컬럼이 없는 브릿지 컬럼을 만들어서 인덱스만으로 조인한다.

이후 조인 결과를 각각의 테이블에 다시 조인해서 결과를 얻는다.

SORT

소트튜닝

SORT MERGE JOIN SORT ORDER BY SORT GROUP BY SORT AGGREGATE SORT UNIQUE (DISTINCT, UNNEST 처리를 위해 UNIQUE 값으로 변경) WINDOW SORT

DISTINCT 와 EXISTS

1 레벨의 테이블에 대한 조인결과만 보기위해 DISTINCT 를 사용하면 처리해야하는 레코드수가 많아진다. 이때 EXISTS 명령어와 서브쿼리를 이용하면 유리하다.

SEMI JOIN

대응되는 레코드를 발견하면 더이상 조인을 진해아지 않는다.

TOP_N query

ROWNUM을 활용해 버퍼를 만드는 레코드의 수를 제한한다.

TO_N 키 알고리즘을 사용하는 ROW_NUM을 사용해라.

ROWNUM을 새롭게 지정한 컬럼을 사용해선 안된다.

STOPKEY

ROWNUM은 <= value (적거나 같다) 로만 비교할수 있다.

쿼리변환

UNNEST

WHERE 절의 서브쿼리를 메인쿼리와 동등한 위치로 올려 조인한다.

조건절 서브쿼리는 Filter 방식으로 풀리는데 unnest 하면 조인이 가능하다.

NL 조인의 배치 i/o, prefetch 기능을 사용할 수 있다.

M 쪽 테이블이 드라이빙 테이블이면 소트 유니크, 필터 테이블이면 세미 조인으로 풀린다.

뷰머징

inline 뷰, 객체화된 뷰를 메인쿼리로 푼다.

실행계획에 view 가 보였다면 뷰머징을 한 것이 아니다. 인라인 뷰를 썼는데 view가 보이지 않았다면 뷰 머징을 한 것이다.

뷰머징을 할 수 없는 경우

집합연산자(union all) CONNECT BY ROWNUM

함수 (윈도우 함수 집계함수 윈도우 함수) GROUP BY, DISTINCT는 뷰머징이 가능하지만 코스트를 따져보고 결정한다.

SELECT /+ FULL(A) PARALLEL(A 2)/ _ FROM EMP A ORDER BY SAL; SELECT _ FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL,NULL,'ALLSTATS LAST'));

조건절 pushing

옵티마이저의 뷰 처리

- 1차적으로 뷰 Merging을 수행하지만 상황에 따라 조건절 PUSING을 시도할 수 있다.

조건절 PUSH DOWN

```
-- 위 QUERY에서 deptno = 30 조건을 인라인 뷰 안으로 밀어 넣어 처리량을 대폭 감소한다.
SELECT /*+PUSH_PRED*/
DEPTNO, AVG_SAL
FROM (
  SELECT DEPTNO, AVG(SAL) AVG_SAL FROM EMP GROUP BY DEPTNO
) A
WHERE DEPTNO = 30;
```

조건절 이행 후 PUSH PRED

```
SELECT E.DEPTNO, D.DNAME, E.AVG_SALS
FROM
(
  SELECT /*+NO_MERGE*/ DEPTNO, AVG(SAL) ALV_SAL
FROM SCOTT EMP
GROUP BY DEPTNO
) E, SCOTT.DEPT D
WHERE E.DEPTNO = D.DEPTNO
AND D.DEPTNO = 30; -- EMP 테이블에도 DEPTNO 컬럼이 존재하므로 뷰테이블로 이행되어 인덱스
스캔을 유발한다.
```

조건절 PULLUP

```
SELECT DEPTNO, AVG_SAL
FROM (
  SELECT DEPTNO, AVG(SAL) AVG_SAL FROM EMP WHERE DEPTNO = 30 GROUP BY
DEPTNO
) E1, -- E1 의 조건절이 pullup 해 E2 로 이행되어 push down 인덱스 스캔을 유발한다.
(SELECT DEPTNO, MIN(SAL)AVG_SAL FROM EMP GROP BY DEPTNO) E2
WHERE E1.DEPTNO = E2.DEPTNO;
```

조인조건 push down

- 조인 조건을 QUERY BLOCK으로 밀어넣는 기능 - 부분범위 처리에 특히 유용한 기능

```
  SELECT *
  FROM DEPT D, EMP E
  WHERE E.JOB = 'CLERK'
  AND D.DEPTNO = 30
  AND D.DEPTNO = E.DEPTNO

  -- 조건절 이행
  SELECT *
  FROM DEPT D, EMP E
  WHERE E.JOB = 'CLERK'
  AND E.DEPTNO = 30
  AND D.DEPTNO = 30
```

불필요한 조인 제거

```
-- 조인시 조인이 실패한 데이터는 안나오기 때문에 조인을 안하는 것과는 다름
select e.empno, e.ename. e,deptno
from dept d, emp e
where d.deptno = e.deptno;
```

dept에 PK 제약조건, emo fk 제약조건이 존재하며 emp.deptno 컬럼에 not null 제약이 존재하면 1쪽 집합은 불필요하다 (항상조인). M쪽이 드라이빙테이블이고 아우터 조인이라면 1쪽 집합은 불필요하다.

OR EXPAND

```
SELECT *
FROM EMP
WHERE SAL = 3000
OR DEPTNO = 200;


-- UNION ALL

SELECT * FROM EMP
WHERE SAL = 3000
UNION ALL
SELECT * FROM EMP
WHERE DEPTNO = 200
AND LNNVL(SAL = 3000)
```

or expand 유도 힌트

- USE_CONCAT - NO EXPAND

기타쿼리변환

```
select count(e.empno), count(d.dname)
from emp e, dept d
where d.deptno = e.deptno
and d.sal <= 290;


select count(e.empno), count(d.dname)
from emp e, dept d
where d.deptno = e.deptno
and d.sal <= 290
and e.deptno is not null
and d.dpetno is not null
```

- Count 시 Null 포함 안됨 - 조건절에 is not null 추가로 결과집합에서 제외 후 Count

비교연산

변별력이 좋은 컬럼이 선두면 비교연산의 횟수가 줄어든다.

파티션

일반적으로 날짜로 테이블을 파티셔닝한다.

파티셔늬 장점은 파티션별로 TRUNCATE 가 가능하다는 것이다.

DELETE 는 DML인 반면 TRUNCATE 는 DDL 이라 빠르다.

PARTITION DROP 또한 DDL로 처리가 빠르다.

고객 과거정보 삭제시 매우 유리하다.

인덱스를 파티션 별로 만들 수 있다 (LOCAL PARTITION INDEX)

이는 인덱스 루트 리프가 파티션의 수 만큼 존재함을 의미한다.

파티션 키가 로컬 파티션 인덱스에 포함되지 않으면 모든 파티셔네 대해 수직적 탐색을 진행한다.

파티션 키가 인덱스에 포함되면 파티션 수직적 탐색의 수를 줄일 수 있다. (PARTITOIN PLUNNG)

파티션 키가 인덱스와 검색조건에 포함되엉 탐색 효율이 좋다. 특히 파티션 테이블이 룩업 테이블일 경우 속도를 보장할 수 없다.

파티션 되지않은 인덱스는 글로벌 파티션 인덱스가 아닌 논파티션 인덱스이다.

논 파티션 인덱스는 파티션 truncate 나 DROP 발생시 REbulid가 필요하다.

파티션 풀스캔으로 유도가 가능하다.

각각의 파티션 테이블은 새그먼트를 가지고 있지않고 다수의 파티션이 하나의 세그먼트에 포함되어있다. 물리적으론 떨어져있고 논리적으론 하나의 세그먼트에 존재한다.

RANGE 파티셔닝

파티션 키 값을 범위로 분할, 주로 날짜 칼럼을 기준

HASH 파티션

파티션 키 값에 해시 하무를적용하고,거기서변환된값으로파티션 매핑

파티션 구성 방법

```
PARTITION BY RANGE(매출일자)
(
  PARTITION P1 VALUES LESS THEN('20190401')
  -- 모든 파티션 키의 기준이 지나서 알맞는 파티션을 찾지 못하면 여기에 적제
  PARTITION P_MAX VALUE LESS THEN (MAXVALUE)
)
```

MAXVALUE는 나중에 기준에 맞게 스플릿해야해서 noN PARTITION INDEX가 깨질 위험성이 있다. 책에선 항상 MAXVALUE를 만들어 장애에 대비해야한라고 기술된다.

PARTITON PLUNNING

옵티마이저가 sQL의 대상 테이블과 조건절ㅇ르 분석한 후 불필요한 파티션을 액세스 대상에서 제외하는 기능

정적

액세스할 파티션을 컴파일 시점에 미리 결정하며, 파티션 키 컬럼을 상수 조건으로 조회

동적

엑세스할 파티션을 실행 시점에 결정, 바인드 변수로 조회하는 경우가 대표적

파티션 키 칼럼에 대한 가공이 발생하면안된다. 데이터 타입의 묵시적 형 변환 시에도 정상적인 파티션 PLUNNING 이 미작동한다.

인덱스 파티셔닝

LOCAL 파티션 vs.GLOBAL 파티션 인덱스

- LOCAL PARTITION INDEX : 테이블 파티션과 1:1로 대응 되도록 파티셔닝한 인덱스 - global partition index : 테이블 파티션과 독립적인 구성을 갖도록 파티셔닝한 인덱스

PREFIXED PARTITION INDEX VS NONPREFIXED PARTITION INDEX

PREFIXED : 파티션 인덱스를 생성할 때, 파티션 키 컬럼을 인덱스 선두 컬럼에 위치

NON : 인덱스 선두컬럼이 아닌 곳에 파티션 위치

파티션 인덱스를 이용하면 sort order by 대체 효과를 상실한다.

비파티션 인덱스는 파티션 이동 삭제 등의 작업시 unusable 되므로 주의해야한다.

파티션 키 컬럼은 주로 일자에 대한 컬럼을 쓰는데 주로 between 조건을 이용하기 때문에 prefixed 로 구성시 인덱스 스켄 효율이 감소할 수 있다.

배치프로그램 튜닝

병렬처리의 활용

병렬은 FULL TABLE SCAN으로 읽어야만 작동한다.

```
-- SINGLE TABLE

select /*+ full(a) parallel(A 2)*/ *
from emp A

-- P001, P002 , QC

-- SELECT STATEMENT
```

```
--  PX COORDINATOR
--    PX SEND QC            -- P - S
--      PX BLOCK ITERATOR   -- 2개 서버로 병렬처리
--        TABLE ACCESS FULL OF EMP
```

PARALLEL SERVER PROCESS가 선언한 병렬처리 수 만큼 생성되고 처리 경과를 QC(SINGLE SERVER PROCESS)에 전달한다.

ORDER BY, GROUP BY 가 포함된 쿼리를 병렬처리하면 분배를 위한 P-P 프로세스가 발생한다.

```
-- SINGLE TABLE ORDER BY
-- PARALLEL DEGREE 의 2배 만큼의 PARALLEL SERVER PROCESS 가 발생
select /*+ full(a) parallel(A 2)*/ *
from emp A
ORDER BY SAL -- 또는 GROUP BY

-- READ(P001, P002), SORT(P003, P004), QC

-- SELECT STATEMENT
--  PX COORDINATOR
--    PX SEND QC (ORDER)          -- P-S
--      SORT ORDER BY
--        PX SEND RECEIVE
--          PX SEND RANGE            -- P-P (RANGE 방식)
--            PX BLOCK ITERATOR   -- 2개 서버로 나눔
--              TABLE ACCESS FULL OF EMP
```

서버 프로세스간에 데이터를 전달하는 것을 분배라고 한다.

분배의 종류

- S-P (SINGLE PROCESS TO PARALLER PROCESS) - P-P (PARALLER PROCESS TO PARALLER PROCESS) - P-S (PARALLEL PROCESS TO SINGLE PROCESS)

분배의 방식(PQ_DISTRIBUTE)

- RANGE - BROADCAST - KEY - HASH - ROUND-ROBIN (INSERT - SELECT)

병렬 조인

FULL PARTITION WISE JOIN

두 테이블 모두 join 컬럼 기준으로 PARTITIONING 되어 있는 경우 같은 범위의 파티션끼리만 하나의 PARALLEL 서버가 조인한다. PQ_DISTRIBUTE 중 PARTITION RANGE 방식 사용 p-p 발생 x 일자컬럼 키의 값의 범위가 다른 파티션 끼리 조인할 확률을 0% 다.

```
-- PARTITION KEY COLUMN : 일자
SELECT *
```

```
FROM T1, T2
WHERE T1.일자 = T2.일자
```

PARTIAL PARTITION WISE JOIN

한쪽은 파티셔닝이 되있고 한쪽은 파티셔닝이 안돼있는 경우 파티션이 돼있는 파티션의 파티션의 키를 기준으로 파티션 안되있는 테이블을 파티션한다. PQ_DISTRIBUTE 중 PARTITION KEY 방식 사용

```
SELECT  /*+
LEADING(D) FULL(E) FULL(D) USE_HASH(E)
PARALLEL (E 2) PARALLEL(D 2)
PQ_DISTRIBUTE(E PARTITION  NONE)
*/
*
FROM DEPT D, EMP E
WHERE E.DEPTNO = D.DEPTNO
```

PQ_DISTRIBUTE 힌트

```
/*+PQ_DISTRIBUTE ( LOOKUP TABLE , DRIVING TABLE 파티션 여부, LOOP UP TABLE 파티션 여부)+/
```

DYNAMIC PARTITION WISE JOIN (HASH)

두 테이블 모두 파티션이 안돼있는 경우 파티셔닝 기준을 해시함수를 이용해 만들어 각 테이블을 파티션하여 병렬처리한다.

두 테이블 모두 HASH PARTITINING 을 진행한다.

HASH 방식으로 분배

```
SELECT /*+
  LEADING(D) FULL(E) FULL(D) PARALLEL(E 2) PARALLEL(D 2)
  PQ_DISTRIBUTE(E HASH HASH)
*/
*
FROM DEPT D, EMP E
WHERE E.DEPTNO = D.DEPTNO

-- PLAN
SELECT STATEMENT
  PX COORDINATOR
    PX SEND QC(RANDOM) -- P - S
      HASH JOIN BUFFERED
        PX RECEIVE
          PX SEND HASH -- P - P
```

```
            PX BLOCK ITERATOR
              TABLE ACCESS FULL OF DEPT
        PX RECEIVE
          PX SEND HASH -- P - P
            PX BLOCK ITERATOR
              TABLE ACCESS FULL OF DEPT
```

DYNAMIC PARTITION WISE JOIN (BROADCAST)

한 테이블은 레코드가 적고 한 테이블은 많으면 많은 쪽 테이블의 파티션에 작은 테이블의 레코드 모두를 분배한다.

BROADCAST DISTRIBUTE 방식으로 분배

```
SELECT /*+
  LEADING(D) FULL(E) FULL(D) PARALLEL(E 2) PARALLEL(D 2)
  PQ_DISTRIBUTE(E BROADCASE NONE)
*/
*
FROM DEPT D, EMP E
WHERE E.DEPTNO = D.DEPTNO

-- PLAN
SELECT STATEMENT
  PX COORDINATOR
    PX SEND QC(RANDOM) -- P - S
      HASH JOIN BUFFERED
        PX RECEIVE
          PX SEND BROADCAST -- P - P
              PX BLOCK ITERATOR
                TABLE ACCESS FULL OF DEPT
          PX BLOCK ITERATOR
            TABLE ACCESS FULL OF DEPT
```

NO_PARALLEL

병렬처리가 안된 테이블은 QC가 처리한다.

S-P 만 발생한다.

특정 배치프로세스 일때 SERVER 프로세스 몇개 발생하는지 파악해야한다.

```
SELECT /*+LEADING(D) FULL(E) FULL(D)
PARALLEL (E 2) NO_PARALLEL(D)
PQ_DISTRIBUTE(E BROADCAST NONE)
*/ *
FROM DEPT D, EMP E
WHERE E.DEPTNO = D.DEPTNO

-- PLAN
```

```
SELECT STATEMENT
  PX COORDINATOR
    PX SEND QC(RANDOM) -- P - S
      HASH JOIN
        BUFFER SORT
          PX RECIEVE
            PX SEND BROADCAST -- S - P BROADCAST
              TABLE FULL ACCESS OF DEPT
        PX BLOCK ITERATOR
          TABLE CEESS FULL OF EMP

SELECT /*+LEADING(D) FULL(E) FULL(D)
PARALLEL (E 2) NO_PARALLEL(D)
PQ_DISTRIBUTE(E HASH HASH)
*/ *
FROM DEPT D, EMP E
WHERE E.DEPTNO = D.DEPTNO

-- PLAN

SELECT STATEMENT
  PX COORDINATOR
    PX SEND QC(RANDOM) -- P - S
      HASH JOIN BUFFERED
        BUFFER SORT
          PX RECIEVE
            PX SEND HASH -- S - P BROADCAST
              TABLE FULL ACCESS OF DEPT
          PX RECEIVE
            PX SEND HASH -- P-P
              PX BLOCK ITERATOR
                TABLE CEESS FULL OF EMP
```

sql 이 수행해야 할 작업 범위를 여러 개의 작은 단위로 나누어 여러 프로세스가 동시처리

여러 프로세스가 동시에 작업하므로 대용량데이터를 처리할 때 극적인 수행속도 절감

parallel_index 힌트를 사용할 때는 반드시 index_ffs 힌트도 함께 사용하는 습관이 필요하다.

옵티마이저가 FULL TABLE SCAN을 선택하면 parallel 힌트는 무시된다.

QC

병렬서버프로세스를 생성하는 역할.

병렬도와 오퍼레이션 종류에 따라 하나 또는 2개의 병렬 서버 집합을 할당한다.

서버 ㅍㄹ로 부터 필요한 만큼 서버 프로세스를 확보 및 부족분에 대한 신규 생성

각 별렬 서버로부터 산출물 통합

최종결과 집합을 사용자에게 전송

스칼라 서브쿼리 수행

PQ_DISTRIBUTE

```
SELECT /*+ ORDERED
USE_HASH(B) USE_NL(C) USE_MERGE(D)
FULL(A) FULL(B) FULL(C) FULL(D)
PARALLEL(A, 16) PARALLEL(B, 16) PARALLEL(C, 16) PARALLEL(D, 16)
PQ_DISTRUBUTE(B, NONE, PARTITION)
PQ_DISTRUBUTE(C, NONE, BROADCAST) -- C는 코드로 작은 테이블
PQ_DISTRUBUTE(D, HASH, HASH)
```

SQL 튜닝 (P 534)

LI CALL IO

library 캐쉬 최적화

APPLICATION CURSOR CACHING

BIND 변수 활용

BIND 변수를 이용해 항상 동일한 SQL을 사용하도록 해서 라이브러리 케시를 공유 가능하게 한다.

BIND 변수를 사용하지 않고 상수를 사용하면 매번 쿼리가 달라져 하드파싱이 일어난다 (LITERAL QUERY)

하드파싱을 최소화한다.

DBMS CALL 최소화

APPLLICATION CURCOR CACHING

ONE SQL 을 사용할 수 없을 때 절차형 SQL에서 CURSOR OPEN과 CURSOR CLOSE 최소화해 LIBRARY CACHE 참조를 줄인다. 이를 통해 소프트 파싱을 줄인다. LIBRARY CACHE에서 데이터를 캐싱해오는 작업은 소프트 파싱이지만 laTCH가 있기 때문에 부담이 된다.

ONE SQL

ONE SQL은 한 번만 수행되므로 1번의 PARSING, 1번의 EXECUTE, (결과 레코드 / 1회 FETCH로 읽어올 수 있는 레코드 수) 만큼의 FETCH 가 발생한다.

ARRAY PROCESSING

FETCH 한번에 한건의 레코드를가져오는 게 아니라 한번에 최대한 많은 레코드를 가져오기 위한 기법이다.

DBMS CALL 종류

PARSE

최적화 APPLICATION CURCOR CACHING

EXECUTE

최적화 ARRAY PROCESSING

FETCH

최적화 ARRAY PROCESSING

I/O 성능 효율화

온라인

수직적 탐색 효율화

수평적 스캔 선택도 향상

인덱스 매칭도 증가 - DRIVING 조건 최대화, = 조건 선두

결합인덱스 우선순위

- 빈번하게 조건으로 사용되는 컬럼 - 동등비교인지 여부 - 좋은 카디널리티 - 소트연산 대체

TABLE RANDOM 엑세스 최소화

WHERE 조건에 있는 컬럼이 모두 인덱스에 존재

BATCH 프로그램 튜닝

- PARLLLEL READ - PARAELLEL WRITE - TEMPPORARY TABLESPACE READ/WRIT - DIRECT PATH READ - DIRECT PATH WRITE - LOB TYPE READ NO CACHE 옵션

ORACLE DBMS SGA 구조

SHARED POOL

- LIBRARY CACHE SQL FULL TEXT AND EXECUTION PLAN

- DICTIONARY CACHE

- DATA BUFFER CACHE - REDO LOG BUFFER

배치프로그램 구현 패턴과 튜닝 방안

절차형 프로그래밍

- APPLICATION 커서를 열고, 루프 내에서 다른 SQL 반복처리 - 반복적인 DBMS CALL 발생 - RANDOM 엑세스 위주 - 동일 엑세스 중복 액세스

병목을 일으키는 SQL을 찾아 I/O 튜닝 - 인덱스를 재구성하고 엑세스 경로 최저화

병렬 프로세스 이용

ARRAY PROCESSING GHKFDYD

최대한 onde sql 사용

ONE SQL 프로그래밍

병목을 일으키는 오퍼레이션 I/O 튜닝 임시 테이블 활용 파티션 활용 병렬처리 활용

---

push_pred

모르면 인라인뷰 풀어서 다시 써야함

아키텍쳐 기반 튜닝 원리

LIBRARY CHACHE DBMS CALL TABLE RANDOM ACCESS IO

오라클 서버는 크게 DATABAS, INSTANCE 두개로 나뉜다.

DATABASE는 DATAFILE, CONTROL FILE, REDO LOG FILE 로 나뉜다.

INSTANCE sms SGA AND BACKGROUND PROCESS

SGA = SHARED POOL, DATA BUFFER CACHE, REDO LOG BUFFER

SHARED POOL = LIBRARY CACHE, DICTIONARY CACHE

BACKGROUND PROCESS = PMON SMON DBWR, LGWR, CKPT

INSTANCE RECOVERY : ROLL FOWARD, ROLL BACK, CHECK POINT

ROLL BACK : SMON PGA RELEASE : PMON DATA BUFFER CACHE => DATA FILE : DBWR REDOLOG BUFFER CACHE => REDO LOG FILE : LGWR MEMORY & DISK SYNC : CKPT SQL FULL TEXT + SQL EXECUTION PLAN : LIBRARRY SQUMMA INFO CACHE : DICTIONARY CACHE

서버 프로세스

사용자 프로세스와 통신하며 각종 명령처리

OPTIMIZER 와 SERVER PROCESS 의 관계

server process 가 optimizer 모듈을 실행시켜 실행계획을 최적화 한다.

BLOCK READ, READ BLOCK SORT, DATA BUFFER CACHE WRITE

I/O의 기본단위

디스크 -> 메모리(input), 메모리 -> 디스트 (output) 의 기본단위는 블록이다.

한 행을 읽으려고 해도 블록을 통체로 메모리에 올렷 ㅓ읽어야한다.

익스텐트

테이블 스페이스로부터 공간을 할당하는 단위 익스텐트 내 블록은 인접한 위치 익스턴트는 특정 세그먼트에 소속 익스텐트 하나 당 8개의 블록이 존재

세그먼트

일반적인 하나의 테이블이 세그먼트다 인덱스 하나가 하나의 세그먼트 파티션 하나가 하나의 세그먼트이다. OS에서 보이면 물리적 안보이면 논리적

데이터파일(물리)와 세그먼트(논리)는 M:M 관계이다.

UNDO

TRASACTION ROLLBACK TRANSACTION (인스턴스) 리커버리 시점의 ROLLBACK 단계

REDO

TRANSACTION ROLLFORWARD TRANSACTION (인스턴스) 리커버리 시점의 ROLLFORWARD 단계

로그파일

- Fast commit : commit 발생시 dBWR가 기동하지 않았지만 REDO LOG를 믿고 빠르게 commit 한다. - WRITE AHED LOGGIN : DB BUFFER CACHE 이전에 redo log에 작성, 즉, DBWR 이전에 LGWR 작동 - LOG FORCE AT COMMIT: COMMIT을 만나면 데이터를 보장하기위해 로그만 기록