AI 연산

SQL 작성 가이드

Ver. 1.0

개 정 이 력

버전	작성일	변경사유[1]	변경내용[2]	작성자	승인자
1.0	2020.08.31	최초작성		변선희	

[1] 변경 사유: 변경 내용이 이전 문서에 대해 최초작성/승인/추가/수정/삭제 중 선택 기입

[2] 변경 내용: 변경이 발생되는 위치와 변경 내용을 자세히 기록(장.절과 변경 내용을 기술한다.)

목 차

1. 개요

1.1 SQL 개발 가이드 의의

본 문서의 목적은 AI 연산 프로젝트 개발에 있어서 개발의 생산성과 효율성을 제고하고, 개발된 결과물의 통일성을 유지하여 향후 유지보수의 편의성을 제공하기 위함이다. 본 문서는 SQL 표준 및 성능을 고려한 SQL 작성 방법에 대한 내용을 포함한다. SQL 문의 표준 기술 방법을 제시하여 개발 프로그램에 적용함으로써 SQL 작성 방법을 표준화하고, 보다 쉽게 SQL 을 이해 할 수 있으며, 결과적으로 프로그램 성능 및 유지보수성을 높이는 효과를 얻을 수 있다.

1.2 범위 및 목적

범 위	목 적
SQL 표준	□ 개발 및 구축 어플리케이션에서 사용하는 SQL 문장의 표준화를 통해서 향후 유지 보수의 용이성, 수행 성능 향상 등의 효과를 기대함 - DBMS 측면에서 SQL 실행을 위한 파싱(Hard Parsing) 부하 감소 및 SQL 문장의 재 사용성을 고려하여 SQL 문장 작성 기본 지침을 정의함 - 인덱스 사용을 저해하는 요소의 사전 제거를 위하여 부가적인 SQL 문장 작성 지침을 정의함
처리 방식	□ SQL 의 수행 속도 보장을 위하여 반드시 준수 및 고려할 사항을 정의하여 어플리케이션의 개발 단계에서부터 적용 가능하도록 유도함 - 인덱스 사용을 위해서 SQL 구문 작성시 지양해야 할 사항 - 조인(Join)의 Driving 집합 결정, 연결 순서 등에 대한 지침 - 프로그램적인 로직의 SQL 처리
처리 범위	□ DBMS 의 성능에 직접적으로 영향을 미치는 DB I/O 의 성능 저하 요소를 미연에 방지할 수 있도록 개발 과정에서 범하지 쉬운 사항에 대하여 지침을 제시함

1.3 방침

본 내용은 업무 내 요구사항에 준하여 수정할 수 있다.

1.4 SQL 작성 표준

1.4.1 기본 작성 지침

□ JOIN 을 지정할 때는 ANSI-SQL 방식으로 기술한다.

□ SQL 문장은 이해하기 쉽고 가독성 좋게 작성한다.

□ 테이블 Alias 는 Naming 규칙에 맞추어 표기한다. (※ <영문 대문자 2 자리: TB> + <주제영역영문명> +<테이블영문명>) 등으로 테이블 명이 길어지는 경우 시작과 끝이 쉽게 이해되도록 Alias 를 지정한다.

□ 라인 변경은 키워드, 테이블, 컬럼에서 하는 것을 원칙으로 하나 가독성 향상을 위해 임의로 라인 변경을 할 수 있다.

□ 단어와 단어 사이, 콤마(,) 다음에는 한 칸을 띄우며 콤마(,)는 줄 앞에 위치 시킨다.

□ 힌트(Hint)는 기본적으로 허용하지 않으나 힌트를 꼭 써야 할 경우는 DBA 와 협의한다.

□ 각종 연산자 (+, -, *, /, =, !=, <, >, <=, >=, <>, IN, LIKE, EXISTS, ...) 앞뒤에 (최소)한 칸을 띄운다.

□ WHERE 절 구문에 있는 컬럼에 대해 인위적인 함수를 사용하지 않는다.

□ 인덱스 사용을 위해 부정형 조건보다 긍정형 조건을 사용하도록 한다.

□ 비교하는 상수 값이 숫자인 경우, 해당 컬럼이 NUMBER 타입인지 확인하고 조인의 연결고리가 되는 컬럼들은 같은 데이터 타입을 사용한다.

□ OR 사용을 최대한 자제하고, IN 등으로 대체될 수 있는지를 검토한다.

1.4.2 라인 변경

□ 라인 변경은 키워드와 컬럼에서 하는 것을 원칙으로 한다.

□ SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, INTO, SET 등과 같이 SQL 구문이 나눠질 경우 라인을 변경한다.

□ AND 의 경우도 역시 라인을 변경한다.

□ 라인이 길어질 경우, 즉, SELECT 조회 컬럼이 많은 경우나 FROM 절 테이블이 많은 경우 개발자 판단에 의해 라인을 변경할 수 있다.

```
SELECT    /* coachingMapper.staticsPageview (코칭통계화면조회) 10012415*/      [필수]
          A1.COL1                    /* 컬럼 1 */          [선택]
          , A1.COL2                  /* 컬럼 2 */
          , SUM(A2.COL2)             /* 컬럼 3 합계 */
FROM      TABLE A1,
LEFT OUTER JOIN
          (
           SELECT      B1.COL1
                      , B2.COL2
           FROM        TABLE B1
           INNER JOIN TABLE B2
           ON   B1.COL1 = B2.COL1
           ) A2
ON        A1.COL1 = A2.COL1
WHERE     A1.COL2 LIKE :emp_id        /*사원 ID*/       [선택]
AND       A2.COL3    = :dept_no       /*부서번호*/
GROUP BY A1.COL1, A1.COL2
ORDER BY A1.COL1, A1.COL2
;
```

1.4.3 들여쓰기

□ SELECT 의 들여쓰기는 SELECT 를 기준으로 FROM, WHERE, INTO 등의 시작점을 맞춘다.

□ SELECT 보다 더 긴 JOIN 절이나, GROUP BY, ORDER BY 절은 해당 절 이후에 스페이스 1 칸을 띄우고 관련 내용을 기술한다.

□ AND 의 경우도 역시 라인을 변경한다.

```
SELECT    /* 프로그램명(한글서비스명) 작성자사번*/          [필수]
          A1.COL1                    /* 컬럼 1 */          [선택]
          , A1.COL2                  /* 컬럼 2 */
          , SUM(A2.COL2)             /* 컬럼 3 합계 */
FROM      TABLE A1,
LEFT OUTER JOIN
          ( /* Subquery 에 대한 설명 */                    [선택]
           SELECT       B1.COL1
                      , B2.COL2
           FROM         TABLE B1
           INNER JOIN TABLE B2
           ON    B1.COL1 = B2.COL1
           ) A2
ON        A1.COL1 = A2.COL1
WHERE     A1.COL2 LIKE :emp_id        /*사원 ID*/          [선택]
AND       A2.COL3     = :dept_no      /*부서번호*/
GROUP BY A1.COL1, A1.COL2
ORDER BY A1.COL1, A1.COL2
;
```

1.4.4 컬럼 라인 변경

□ 장문의 컬럼, IF 나 CASE 문과 같이 가독성이 떨어질 경우 적절히 라인을 변경하거나 스페이스를 추가하여 가독성을 향상시킨다.

□ SQL 구문은 가로 최대 길이가 100 컬럼 이하가 되도록 작성하며, 100 자를 넘기는 경우 라인을 변경할 수 있다.

```
SELECT    /* 프로그램명(한글서비스명) 작성자사번 */          [필수]
          A1.COL1
         , CASE WHEN A1.COL1 = '9801' THEN 'COL1-OK'
                WHEN A1.COL1 = '9802' THEN 'COL2-OK'
                ELSE  'COL3-NO'
          END M1
         , CASE WHEN A1.COL1 = '9801' THEN '1' END M2
         , CASE WHEN A1.COL1 = '9802' THEN '2' END M3
FROM      TABLE_NAME A1
WHERE     A1.COL4 = :dept_no       /*부서번호*/              [선택]
GROUP BY A1.COL1, A1.COL2
ORDER BY A1.COL1, A1.COL2
;
```

1.4.5 주석

□ 주석(Comment)은 복잡한 SQL 문에 대한 설명과 수정 이력 내용을 기술한다

□ 주석은 /* 와 */ 를 사용하고 한 줄 Comment 명령어인 -- (2 개의 연속 Hyphen 기호)은 사용하지 않는다.

□ 작성되는 모든 SQL 문에는 향후 SQL Tuning 및 검색에 대한 편의를 위해 /* 프로그램명(한글서비스명) */ 형태의 Comment 를 기술한다.

```
SELECT    /* 프로그램명(한글서비스명) 작성자사번*/          [필수]
          A1.COL1                    /* 컬럼 1 */          [선택]
         , A1.COL2                   /* 컬럼 2 */
         , SUM(A2.COL2)              /* 컬럼 3 합계 */
FROM      TABLE A1,
LEFT OUTER JOIN
          ( /* Subquery 에 대한 설명 */
           SELECT       B1.COL1
                      , B2.COL2
           FROM         TABLE B1
           INNER JOIN TABLE B2
           ON   B1.COL1 = B2.COL1
           ) A2
ON        A1.COL1 = A2.COL1
WHERE     A1.COL2 LIKE :emp_id        /*사원 ID*/
AND       A2.COL3    = :dept_no       /*부서번호*/
GROUP BY A1.COL1, A1.COL2
ORDER BY A1.COL1, A1.COL2
;
```

1.4.6 괄호

□ 인라인 뷰, 스칼라 서브쿼리 등을 사용할 경우 괄호 내에서 위의 작성 표준을 준수한다.

□ 함수 괄호에는 공백을 넣지 않는다.

□ 한 줄이 길어지는 경우에는 개발자의 판단 하에 적절히 라인을 변경하고 가독성이 떨어지지 않도록 유의한다.

□ 인라인 뷰는 시작 괄호'(' 다음 공백 한 칸 뒤에 SELECT 문이 시작되고 문장이 끝난 후 새로운 라인에 시작 괄호와 ')'를 정렬한다.

□ 인라인 뷰/서브쿼리에서 시작 괄호 "(" 다음에 서브쿼리 내 시작점을 맞춘다.

```
SELECT    /* 프로그램명(한글서비스명) 작성자사번*/          [필수]
          A1.COL1                    /* 컬럼 1 */          [선택]
         , A1.COL2                   /* 컬럼 2 */
         , SUM(A2.COL2)              /* 컬럼 3 합계 */
FROM      TABLE A1,
LEFT OUTER JOIN
          ( /* Subquery 에 대한 설명 */                     [선택]
           SELECT      B1.COL1
                     , B2.COL2
           FROM        TABLE B1
           INNER JOIN TABLE B2
           ON   B1.COL1 = B2.COL1
           ) A2
ON        A1.COL1 = A2.COL1
WHERE     A1.COL2 LIKE :emp_id        /*사원 ID*/          [선택]
AND       A2.COL3    = :dept_no       /*부서번호*/
GROUP BY A1.COL1, A1.COL2
ORDER BY A1.COL1, A1.COL2
;
```

1.5 SQL 작성 가이드

1.5.1 SQL 구문작성 예제

1.5.1.1 ANSI 표준을 사용

□ Outer Join 은 사용을 자제하지만 필요 시 ANSI 표준을 사용한다.

□ Outer Join 의 경우 LEFT OUTER JOIN 을 사용하도록 한다. Outer 되는 테이블의 조건은 반드시 on 절 다음에 기술하며 Where 절에 기술해서는 안 된다.

```
SELECT    /* 프로그램명(한글서비스명) 작성자사번*/
          A1.COL1                    /* 컬럼 1 */
          , A1.COL2                  /* 컬럼 2 */
          , SUM(A2.COL2)             /* 컬럼 3 합계 */
FROM      TABLE A1,
LEFT OUTER JOIN
          ( /* Subquery 에 대한 설명 */
           SELECT       B1.COL1
                      , B2.COL2
           FROM         TABLE B1
           INNER JOIN TABLE B2
           ON    B1.COL1 = B2.COL1
           ) A2
ON        A1.COL1 = A2.COL1
WHERE     A1.COL2 LIKE :emp_id        /*사원 ID*/
AND       A2.COL3     = :dept_no      /*부서번호*/
GROUP BY A1.COL1, A1.COL2
ORDER BY A1.COL1, A1.COL2
;
```

1.5.1.2 SELECT

□ SELECT 를 기준으로 FROM, WHERE, AND, OR, ORDER BY, GROUP BY, HAVING, INTO, UNION ALL 등의 시작점을 맞춘다.

□ 테이블/컬럼 Alias 가 한번 정의된 후에 그 테이블/컬럼을 참조하는 할 때 반드시 Alias 를 사용한다.

□ 한 라인에 한 컬럼만 표기하며 모든 컬럼을 뜻하는 * (Asterisk)는 사용하지 않는다.
* 를 사용할 경우 해당 테이블의 모든 컬럼을 fetch 하여 핸들링 하기 때문에 메모리 낭비를 초래 할 수 있다.

□ 컬럼은 첫 번째만 제외하고 콤마로 시작하고 콤마 뒤에 한 칸 띄운 후 컬럼을 기술하며, 필요 시 주석을 컬럼 뒤에 기술한다.

□ SELECT 를 통해 조회하는 컬럼은 주석(/* */)으로 컬럼명 또는 컬럼 내용을 기술한다.

```
SELECT    /*+ 프로그램명(한글서비스명) 작성자사번 */
          A1.COL1                    /* 컬럼 1 */
         , A1.COL2                   /* 컬럼 2 */
         , SUM(A2.COL2)        /* 컬럼 3 합계 */
FROM      TABLE A1,
LEFT OUTER JOIN
          ( /* Subquery 에 대한 설명 */
           SELECT       B1.COL1    [컬럼에 테이블 ALIAS 사용]
                      , B2.COL2
           FROM         TABLE B1
           INNER JOIN TABLE B2
           ON    B1.COL1 = B2.COL1
           ) A2
ON        A1.COL1 = A2.COL1
WHERE     A1.COL2 LIKE :emp_id          /*사원 ID*/
AND       A2.COL3     = :dept_no        /*부서번호*/
GROUP BY A1.COL1, A1.COL2
ORDER BY A1.COL1, A1.COL2
;
```

1.5.1.3 FROM

□ 테이블이 하나일 경우에는 Alias 를 지정하지 않으며 2 개 이상의 테이블이 사용될 때는 반드시 Alias 를 사용한다.

□ Alias 는 알파벳 대문자 1 자리 + 일련번호로 구성하고 한 SQL 문장에 중복 사용하지 않는다. <영문 한 문자> + <일련번호>

- 외부 테이블: A1, A2, A3, ...
- 첫 번째 깊이의 INNER 테이블: B1, B2, B3, ... → 동일 레벨의 블록일 경우 위에서 순서대로 일련번호 작성

□ 인라인 뷰는 시작 괄호 '(' 다음 공백 한 칸 뒤에 SELECT 문이 시작되고 문장이 끝난 후 새로운 라인에 시작 괄호와 ')'를 정렬한다.

□ JOIN 조건절은 FROM 절과 WHERE 절 사이에 JOIN 구문의 ON 절에 작성한다.

□ 한 라인에는 하나의 테이블 명만 기술하고 주석으로 테이블을 설명한다.

□ Driving 순서대로 테이블을 순차적으로 기술하여 Driving 하려는 테이블의 의도를 쉽게 이해할 수 있도록 한다.

```
SELECT    /* 프로그램명(한글서비스명) 작성자사번 */
          A1.COL1                    /* 컬럼 1 */
         , A1.COL2                   /* 컬럼 2 */
         , SUM(A2.COL2)              /* 컬럼 3 합계 */
FROM      TABLE A1,
LEFT OUTER JOIN
          ( /* Subquery 에 대한 설명 */
           SELECT      B1.COL1
                      , B2.COL2
           FROM        TABLE B1   [첫 번째 깊이의 INNER 테이블]
           INNER JOIN TABLE B2
           ON   B1.COL1 = B2.COL1
           ) A2   [인라인뷰 ALIAS 사용]
ON        A1.COL1 = A2.COL1   [JOIN 조건절 삽입(WHERE 절 앞부분]
WHERE     A1.COL2 LIKE :emp_id        /*사원 ID*/
AND       A2.COL3    = :dept_no       /*부서번호*/
GROUP BY A1.COL1, A1.COL2
ORDER BY A1.COL1, A1.COL2
;
```

1.5.1.4 WHERE

□ 한 라인에 하나의 조건절만 기술하고 AND, OR 는 새로운 라인에 기술한다.

□ 조건절은 산술연산자 (+, -, *, /, ||, =, !=, <, >, <=, >=, <>)의 오른쪽을 기준으로 정렬하고, 비교연산자(IN, LIKE, EXISTS)는 주변 상황에 따라 가독성 좋게 정렬한다.

□ 조건절은 연산자 조건, 상수 조건 및 범위 조건, 변수 조건, 서브쿼리 순서대로 기술한다. (먼저 처리되는 처리 범위부터 순서대로 기술한다.)

□ 서브쿼리는 시작 괄호 '(' 다음 SELECT 문이 시작되고 문장이 끝난 후 해당 라인에 ')'를 통해 종료한다.

□ 성능을 고려하여 WHERE 절에 인덱스가 걸려있는 컬럼(인덱스 컬럼)을 변경하지 않도록 하도록 한다.

□ ORDER BY 나 GROUP BY 는 WHERE 조건이 아닌 별도 라인에 기술한다.

```
SELECT      /* 프로그램명(한글서비스명) 작성자사번 */
            A1.COL1                    /* 컬럼 1 */
           , SUM(A2.COL2)
FROM        TABLE A1,
INNER JOIN TABLE A2
ON         A1.COL1 = A2.COL1
WHERE      A1.COL2   >= A2.COL2   [조건절 연산자 정렬]
AND        A2.COL2 LIKE 'ABC%'    [상수 조건]
AND        A2.COL3      = :col4       [변수 조건]
AND        A1.COL4 IN (SELECT    B1.COL1
                       FROM      TABLE3 B1
                       WHERE     B1.COL1 IN ('01','02'))
AND        EXISTS (SELECT 'x'
                   FROM     TABLE4 B2
                   WHERE    B2.COL1= A1.COL1)
;
```

1.5.1.5 INSERT

□ INTO, VALUES, SELECT, FROM, WHERE, AND, OR 등은 INSERT 의 시작라인에 맞춘다

□ 한 라인에 하나의 컬럼/값만 기술하고 콤마(,)는 컬럼은 첫 번째만 제외하고 콤마로 시작한다.

```
INSERT   /* 프로그램명(한글서비스명) 작성자사번 */          [필수]
INTO     TABLE1 A1
         ( A1.COL1       /* Comment */                    [선택]
         , A1.COL2       /* Comment */
         , A1.COL3 )     /* Comment */
SELECT   A2.COL1
         , A2.COL2      /* Comment */
         , SYSDATE
FROM     TABLE2 A2
WHERE    A2.COL1 = '111'
;
```

1.5.1.6 DELETE

□ FROM, WHERE, AND, OR 등은 DELETE 의 시작라인에 맞춘다.

□ FROM 은 생략하지 않고 반드시 표기한다.

```
DELETE    /* 프로그램명(한글서비스명) 작성자사번*/        [필수]
FROM      TABLE1
WHERE     A1.COL5 LIKE 'ABC%'    /* Comment */          [선택]
AND       A1.COL6    = :col6
AND       A1.COL7 IN (SELECT    COL1
                      FROM      TABLE3
                      WHERE     COL2 = '001')
;
```

1.5.1.7 UPDATE

□ SET, FROM, WHERE, AND, OR, 콤마(,)등은 UPDATE 의 시작 라인에 맞춘다.
□ 한 라인에 하나의 업데이트될 컬럼만 기술한다..

```
UPDATE    /* 프로그램명(한글서비스명) 작성자사번 */ TABLE1 A1
SET       A1.COL1 = :col1,   /* Comment */
          (A1.COL2, A1.COL3) =   (SELECT    B1.COL2
                                          , B1.COL3
                                  FROM      TABLE2 B1
                                  WHERE     B1.COL1 = '111'),
           A1.COL4 = TO_CHAR(SYSDATE,'yyyy-mm-dd hh24:mi:ss')
WHERE     A1.COL5 LIKE 'ABC%'
AND       A1.COL6 = :col6
AND       A1.COL7 IN (SELECT COL1
                      FROM   TABLE3
                      WHERE COL2 = '001')
;
```

1.5.2 성능 향상을 위한 부가적 작성 지침

□ [3]SQL Parsing 부하 감소를 위해 재사용 가능한 바인드 변수를 사용한 문장으로 작성한다.

□ 조회는 가급적 PK 기반으로 요청해서 row lock 레벨로만 잡히도록 구현한다.

□ 트랜잭션이 길게 걸리는 작업은 가급적 지양한다.

□ 기본 작성 지침의 대소문자 및 단어 사이 간격 유지 방법 준수한다.

□ 불필요한 IFNULL 함수를 사용하지 않는다. 필요 시 해당 컬럼에 기본(Default)값을 지정하여 사용하도록 한다.

□ 만약 FUNCTION 을 SQL 의 컬럼 레벨에서 사용하고자 할 때, 반드시 최종결과 RECORD 에서 사용한다. 즉 FUNCTION 의 호출 횟수를 줄이도록 한다.
- 인라인 뷰에서 사용되면 불필요한 함수 호출로 DB 성능에 악영향을 준다.

□ WHERE 절에 부정형 조건보다 긍정형 조건을 사용한다.

□ OR 사용을 가급적 제한해서 사용한다.

□ 인덱스 컬럼의 변형을 피하기 위해서 SUBSTR(COL1,1,2) = 'AB' 등과 같은 구분은 COL1 LIKE 'AB%'와 같이 사용한다.

□ 만약 일어날 수 있는 특정 수를 알고 있다면, LIKE 를 사용하는 것보다 IN 에 목록을 나열한다.

□ NOT 의 사용(EXISTS 는 제외)을 피하며, NOT 은 단지 매우 복잡한 조건에서만 선택적으로 사용한다.

□ <=, >= 보다는 BETWEEN 을 사용한다.

□ 인라인 뷰의 SELECT-LIST 에 있는 칼럼 중에서 MAIN-QUERY 와 조인되는 칼럼이 WHERE 절에 사용되는 칼럼은 SUB-QUERY 에서 변형하지 말고 MAIN-QUERY 에서 칼럼을 변경하도록 한다.

[3] SQL Parsing: SQL 문장에 문법적 오류가 없는지(Syntax 검사), 의미상 오류가 없는지(Semantic 검사) 등을 체크하는 일련의 과정

1.5.2.1 바인드 변수의 사용

□ 하드 파싱[4](Hard Parsing) / 소프트 파싱[5](Soft Parsing)

- SQL 문이 대소문자, 띄어쓰기, 공백, 길이, 주석 등 모든 것이 같을 때만 동일한 SQL 문장으로 간주된다.
- 하드 파싱은 바인딩 변수 미사용, 프로그램상의 SQL 조립에 의해 발생하기 쉽다.

□ 실행 시 매번 하드 파싱을 유발하는 애플리케이션의 문제점

- SQL 파싱 오버헤드, 해킹공격(SQL Injection)에 취약 (바인딩 변수 미사용)
- 조건에 따라 SQL 이 바뀌므로 튜닝 불가
- Shared Pool 메모리의 단편화를 유발시키며 심할 경우 데이터베이스 시스템이 다운된다.

□ Parsing 부하 감소를 위해 재사용 가능한 SQL 문장을 작성한다.

- SQL 문에 상수 값을 사용하여 작성할 때 서로 다른 상수 값은 Parsing 시 다른 SQL 문으로 인식하여 재 Parsing 하므로 Bind Variable 을 사용하여 같은 SQL 이 공유될 수 있도록 함
- Parsing 은 CPU 를 많이 소모하므로 Parsing 을 피할 수 있도록 SQL 이 공유되게 작성함
- 문장 ①과 ②는 다른 SQL 문으로 인식되어 재 Parsing 되며, ③과 ④는 서로 같은 문장이므로 재 Parsing 하지 않음.

```
① SELECT NAME FROM EMP WHERE NUM = 9;
② SELECT NAME FROM EMP WHERE NUM = 10;
③ SELECT NAME FROM EMP WHERE NUM = :num;
④ SELECT NAME FROM EMP WHERE NUM = :num;
```

1.5.2.2 재사용성

□ 기본 작성 지침의 대소문자 및 단어 사이 간격 유지 방법 준수

- 기본 작성 규칙을 지키지 않는 경우 결과가 같은 SQL 문이라도 SQL 문이 공유되지 않아 Parsing 이 발생하여 CPU 를 소모하게 됨
- 문장 ①과 ②는 같은 결과를 주는 문장이지만, 두 문장을 구성하는 Word 의 간격이 다르므로 다른 SQL 로 인식되어 재 Parsing 됨

[4] 하드파싱 : 공유 메모리에서 SQL 을 찾기 못해 실행계획을 새로 생성하고 실행하는 것.

[5] 소프트파싱 : 공유 메모리에서 SQL 과 실행계획을 찾아 바로 실행하는 것

```
① SELECT NAME FROM EMP WHERE NUM = :num;
② SELECT NAME FROM EMP WHERE NUM = :num;
```

- ③과 ④도 같은 결과를 주는 문장이지만, 대소문자가 다르면 서로 다른 SQL 로 인식되어 재 Parsing 됨

```
③ SELECT NAME FROM EMP WHERE NUM = :num;
④ select name from emp where num=:num;
```

2. 처리 방식 가이드라인

2.1 SQL 튜닝 기본지침

□ SQL 언어를 정확히 이해하고 작성해야 한다.
- JOIN 의 종류 및 특성에 대한 정확한 이해를 바탕으로 SQL 을 작성해야 한다.
- NOT IN 과 NOT EXISTS 의 정확한 차이점, IN 과 EXISTS 의 정확한 차이점을 알고 작성해야 한다.

□ SQL 구문 이해를 돕기 위하여 복잡한 In-Line View 의 사용보다는 테이블 조인을 하도록 한다.

□ 불필요한 DUAL 테이블의 사용은 제한한다.

□ Distinct 는 항상 Sort 연산을 수행하므로, 처리결과값이 항상 unique 한 경우 DISTINCT 키워드를 사용하지 않는다.

□ 서브 쿼리는 M:M 관계에 있는 데이터 모델의 연결에서 어느 한쪽 집합이 단순히 조건 체크로 사용되거나, Main Query 조건에 상수 값을 제공하는 경우에만 사용하도록 제한한다.
- 서브 쿼리는 메인 쿼리에 대해 제공자, 확인자 역할을 담당한다.
- 제공자: 메인 쿼리보다 먼저 수행되어 수행 결과값을 제공. 주로 IN 형태의 서브 쿼리
- 확인자: 메인 쿼리가 먼저 수행되어 메인 쿼리의 결과값에 대한 체크 역할을 담당. 주로 EXISTS 형태의 서브 쿼리
- 서브 쿼리의 결과값이 중복된 값이 있을 경우 유일한 값으로 먼저 추출되기 때문에 DISTINCT 및 의미 없는 그룹 함수 사용을 제한한다.

□ WHERE 절 구문에 있는 컬럼에 대하여 인위적인 함수를 사용하지 않는다.

□ 인덱스 사용을 위해 부정형 조건보다 긍정형 조건을 사용하도록 한다.

□ 서로 다른 data 타입 비교 시 DBMS 가 한쪽을 기준으로 내부 변형을 발생 시키므로 주의한다.

□ 비교하는 상수 값이 숫자인 경우, 해당 컬럼이 number 타입인지 확인하고, 조인의 연결고리가 되는 컬럼들은 같은 데이터 타입 사용한다.

□ INLINE VIEW 를 적절히 활용하여 조인 횟수를 줄이거나 SQL 호출 수를 줄일 수 있다.

□ 결합의 선행 컬럼을 검색 조건에 포함한다.

□ NULL 비교 조건은 피하고, 빈번히 검색 대상이 된다면 'NULL' 대신 Default Value 를 정하고 SQL 을 수정한다.

2.2 인덱스

□ SQL 을 기술할 때 적절한 인덱스를 사용토록 하는 것은 table full scan 등으로 인한 성능저하를 막고 시스템의 부하를 줄일 수 있는 방법이기 때문이다. 따라서, 적절한 인덱스를 만들고 인덱스를 사용토록 구문을 작성하는 것은 프로젝트 전반에 걸친 성능향상 작업의 시초가 된다.

□ 테이블 내의 한 개 또는 여러 개의 칼럼 값과 그 값을 포함하고 있는 행(ROW)의 논리적 주소(RowID)를 연관시켜 만들어지는 별도의 저장 구조로 테이블에 대해 빠른 액세스 경로를 제공하여 SQL 문의 실행 속도를 빠르게 하고 디스크 I/O 를 줄일 수 있다.

2.2.1 인덱스 생성 기준

□ 인덱스는 테이블의 특성, 컬럼의 분포도, 처리 범위 등을 정확히 파악하여 생성해야 함. 특정 어플리케이션에만 영향을 미치는 것이 아니라 그 컬럼을 사용하는 모든 경우에 영향을 미치므로 액세스 빈도, 처리 범위의 크기, 분포도 등을 감안하여 종합적으로 결정해야 함.

□ where 절에서 "=", "like", "between" 등을 사용하는 컬럼

□ 컬럼 변형이 있는 경우 제외하지만 변형 없는 SQL 문으로 재 작성이 가능하다면 가능함

□ where 절에서 부정형(not, <>), null 비교 등을 사용하는 컬럼은 제외하지만 부정형이나 null 비교를 없앨 수 있다면 가능함

□ 컬럼의 data 분포도가 10~15% 이내의 경우 사용

- 분포도=(1/컬럼 값의 종류) *100=(컬럼 값의 평균 로우 수/테이블의 총 로우 수)*100

□ 하나의 테이블에 6 개 이상의 인덱스 생성은 온라인 업무처리 시 오버헤드가 발생하므로 6 개 이상의 인덱스에 대해서는 DBA 와 협의하여 생성

□ 신규 인덱스 생성은 기존 수행 쿼리의 Access Path 에 영향을 줄 수 있으므로 고려하여 생성

□ 여러 개의 인덱스(컬럼 1 개)보다 한 개의 결합 인덱스(다수 컬럼)로 생성하는 게 유리

2.2.2 결합 인덱스 생성 조건

□ 한 개 테이블에 여러 개 인덱스가 있더라도 일반적으로 하나의 인덱스만 사용됨으로 두 개 이상의 컬럼이 조인 조건에 사용되는 경우는 결합 인덱스를 사용

□ 결합 인덱스의 구성된 컬럼들 중 반드시 첫 번째 컬럼은 index 를 사용할 수 있는 조건이 되어야 함

□ where 절에 여러 개의 상수 조건을 갖는 경우 한 컬럼에 인덱스가 있는 것보다 상수 조건을 갖는 컬럼들을 하나 인덱스로 만들 때 가장 효율적임

□ 결합 인덱스 leading 컬럼 가이드라인

- 결합인덱스의 첫 번째 컬럼을 조건에서 사용하지 않으면 그 인덱스는 사용되지 않으며 처리 범위를 결정하는 데에도 큰 영향을 미침. 따라서, 첫 번째 컬럼을 결정하는데 있어서 가장 중요한 기준은 그 조건에 항상 사용되는 컬럼들 중에서 선정되어야 함
- 인덱스를 생성하려는 대상 컬럼 중 분포도가 가장 좋은 컬럼을 leading 컬럼으로 선택해야 함

2.2.3 인덱스를 사용하지 못하는 SQL 문

2.2.3.1 인덱스를 사용 못하는 유형

□ 인덱스 컬럼의 내외부 변형, Not 연산자, Null Or Not Null Value 의 비교, DBMS Optimizer 의 취사선택에 의하여 인덱스 사용을 저해하는 요소는 SQL 문장 작성시 지양하여야 함.

INDEX COLUMN의 변형	SELECT * FROM DEPT WHERE SUBSTR(DNAME,1,3)='ABC'
NOT Operator	SELECT * FROM EMP WHERE JOB <> 'SALES'
NULL, NOT NULL	SELECT * FROM DEPT WHERE ENAME IS NOT NULL
Optimizer의 취사선택	SELECT * FROM DEPT WHERE JOB LIKE 'AB%'

2.2.3.2 인덱스 컬럼의 외부(External) 변형

□ 사용자가 인덱스를 가진 컬럼을 SQL 함수나 사용자 지정함수, 연산, 결합(||) 등으로 가공시킨 후 비교할 때 발생함.

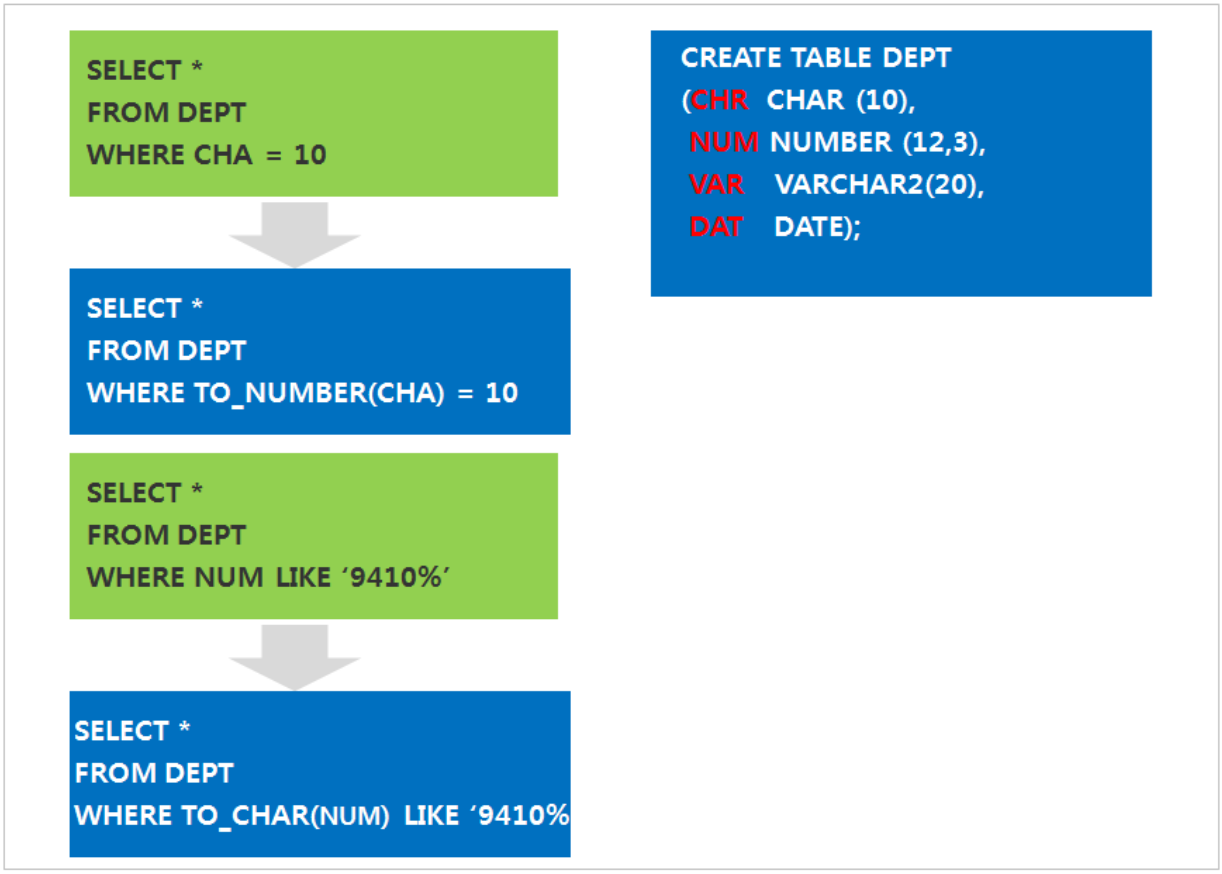
□ 인덱스 설계가 완료되지 않은 개발 초기 단계부터 컬럼에 대한 변형 지양.
개발 완료시점에 인덱스가 적용되어도 인덱스를 사용하지 못함.

□ 속성을 통합하여 단일 컬럼으로 물리 설계를 하는 경우 데이터 Access Pattern 을 면밀하게 분석할 필요가 있음.

2.2.3.3 인덱스 컬럼의 내부(Internal)변형

□ 서로 다른 데이터 타입을 비교하고자 할 때 DBMS 가 어느 한 쪽을 기준으로 동일한 타입이 되도록 내부적인 변형을 일으키게 되어 발생함.

□ 내부적인 변형은 의도하지 않은 액세스 경로를 설정하여 수행 속도에 막대한 영향을 미치게 되므로 조건 절에서 비교하는 데이터 값과 컬럼의 데이터 타입을 반드시 동일하게 처리해야 함.



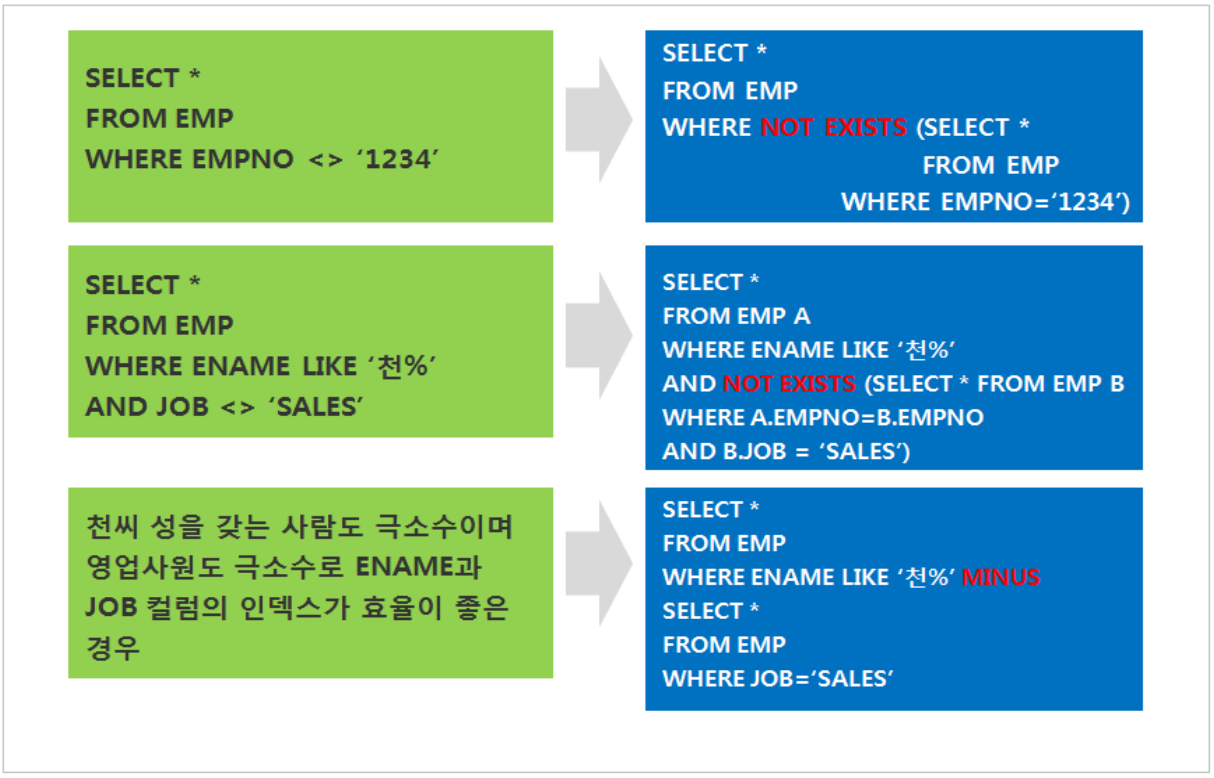
2.2.3.4 NOT Operator

□ 부정형 (NOT, <>, !=) 으로 조건을 기술한 경우에는 인덱스가 사용되지 않으므로 되도록 긍정형으로 바꾸어 인덱스를 사용하도록 유도해야 함.

□ 조회 대상 집합에 대해서 부분범위 처리할 수 있으며 인덱스의 사용이 가능한 'NOT EXISTS'를 활용.

□ 조회 조건에 부합되는 모집합을 조회 시 확인자(체크) 역할을 수행.

□ IS (NOT) NULL 로 비교된 경우 INDEX 에는 NULL 에 대한 정보가 없으므로 Full Table Scan 이 발생하게 됨. CHAR 형의 경우 컬럼명 >' ' 를 사용하여 해당 컬럼의 INDEX 가 사용되도록 할 수 있음.

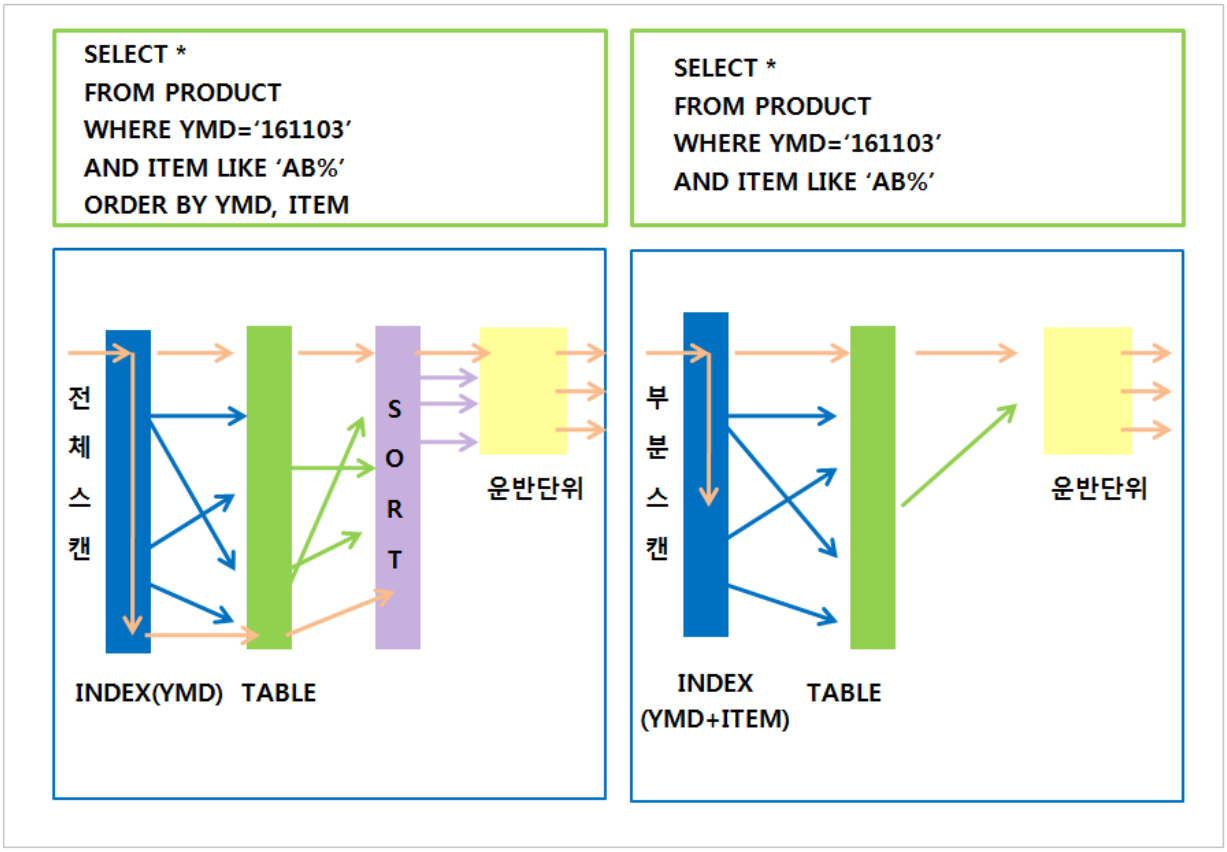


2.3 성능을 고려한 SQL 작성

2.3.1 Sort 를 대신하는 INDEX

□ 조건절의 비교 컬럼과 동일한 인덱스가 있다면 별도의 정렬 작업이 필요 없음.

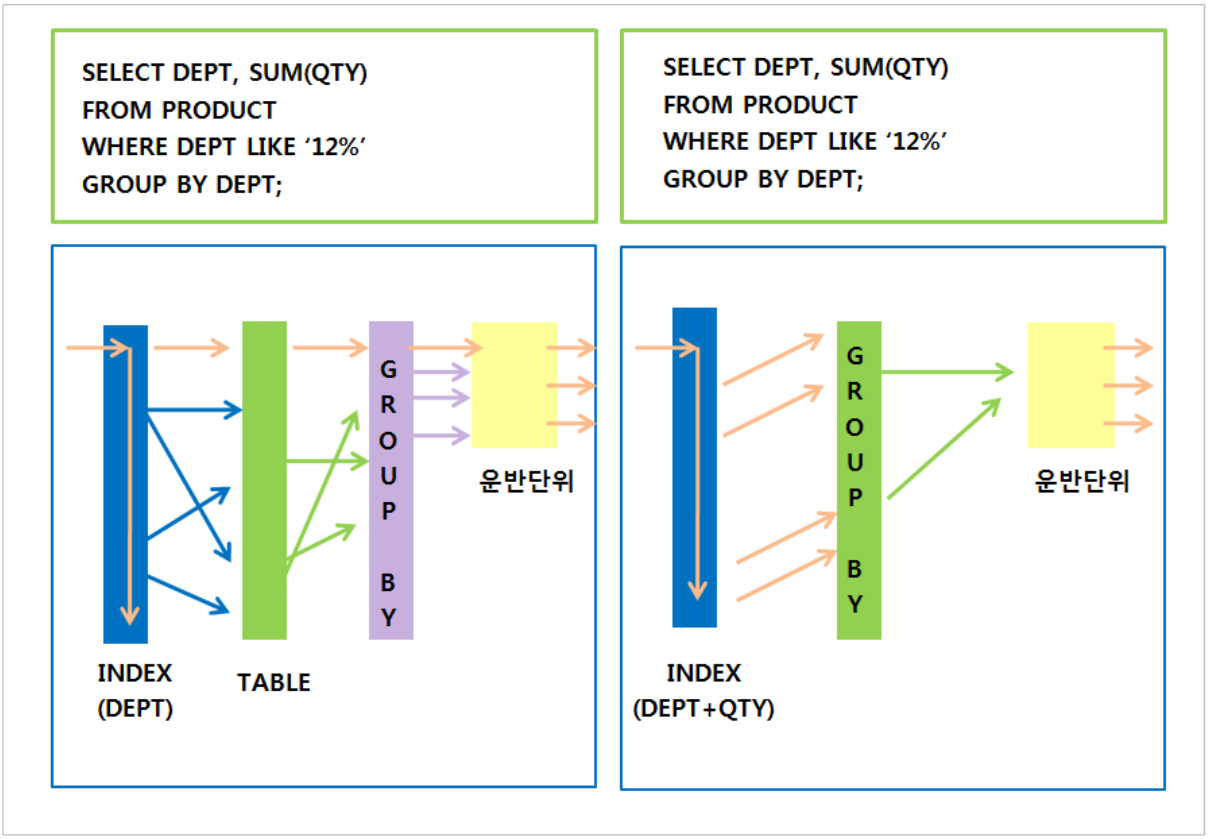
□ ORDER BY 는 전체 범위 처리를 수행하므로 부분범위 처리를 위해서는 조건절 및 정렬 순서에 적합한 인덱스가 존재하여야 함.



2.3.2 INDEX 만 처리

□ 물리적인 Random I/O 를 줄이기 위하여 전략적으로 인덱스 설계를 함.

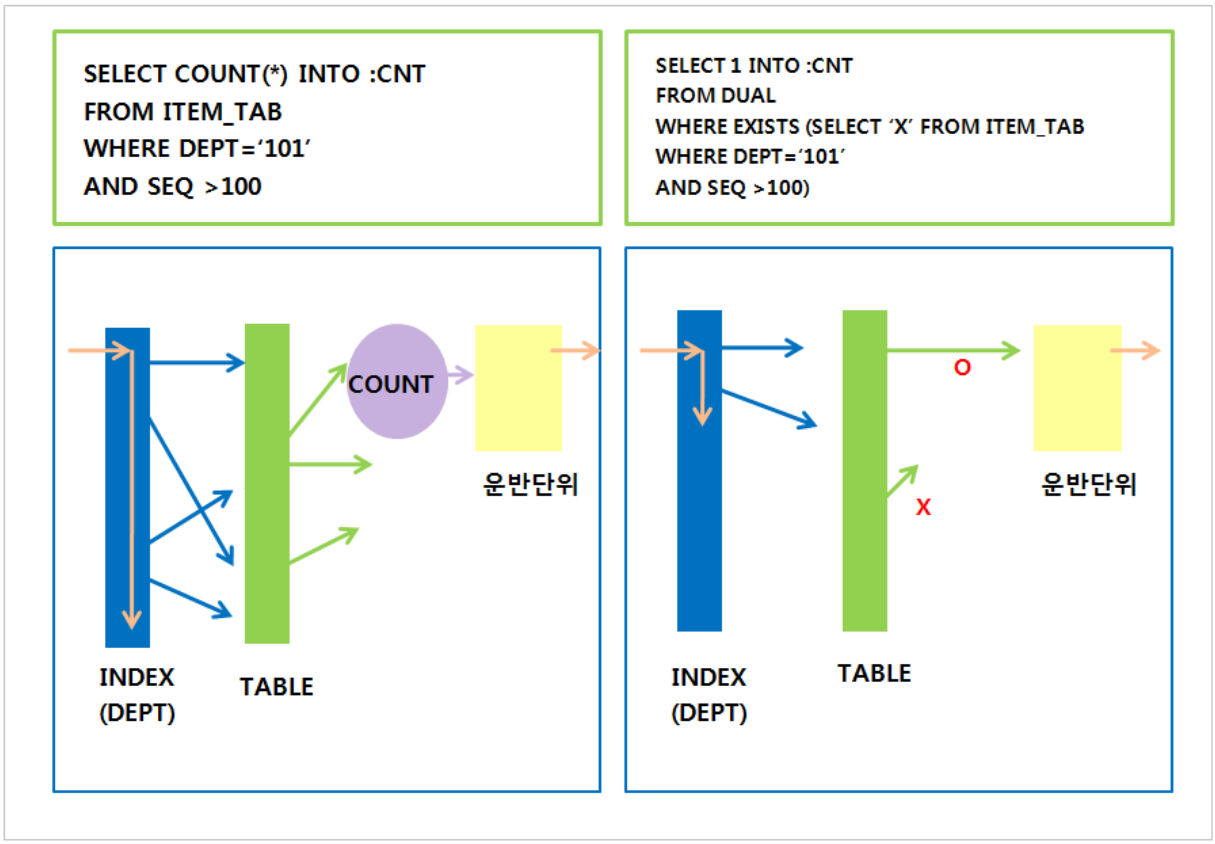
□ 인덱스만을 읽어서 필요한 결과를 조회함으로써 SQL 의 수행 성능을 향상함.



2.3.3 EXISTS 활용

□ 조건에 부합되는 데이터의 존재 유무 확인 시 활용(Count 는 전체 데이터를 Access).

□ EXISTS, NOT EXISTS 는 Boolean(True Or False)의 개념.



2.3.4 Bulk Insert

□ PostgreSQL 에서 여러 개의 Insert 구문 수행 시 Values 리스트를 다중으로 사용하도록 한다.

기존

INSERT INTO TB_A VALUES (A,B,C)

INSERT INTO TB_A VALUES (D,E,F)

변경

INSERT INTO TB_A VALUES (A,B,C),(D,E,F)

□ 인덱스가 많이 걸려있는 테이블에 대량으로 INSERT 시에는 매 ROW 마다 계산 후 데이터가 삽입되므로 부하가 발생되므로 사전에 고려하도록 한다.

□ Insert..Select 형태의 쿼리는 Select 되는 대상을 고정시켜야 하므로 read-lock 을 잡게 되므로 유의하여 사용해야 한다.

2.3.5 실행계획 확인

실행계획 조회 명령어

-explain 쿼리

```
explain
select * from brs.tb_cmn_qst;
```

-explain analyze

```
explain analyze
select * from brs.tb_cmn_qst;
```

analyze 옵션을 사용하면, 실제 해당 쿼리를 실행하고, 추청 비용과 함께 소요 비용, 소요 시간도 실제 처리된 각 계획 노드별 전체 로우 수도 보여준다. 이 옵션은 실행계획기가 추정하는 작업이 실 작업과 비교해서 얼마나 정확한지를 확인하는데 유용하게 사용된다.

중요: ANALYZE 옵션을 사용하면, 지정한 그 쿼리문이 실제로 실행된다는 사실을 꼭 기억하고 있어야 한다. SELECT 구문에 대한 EXPLAIN 작업이 아무런 출력 로우를 보여주지 않지만, 그 작업은 실제 실행되었기 때문에, 다른 어떤 부수적 영향을 다른 부분에 끼칠 가능성이 있다. INSERT, UPDATE, DELETE, CREATE TABLE AS, EXECUTE 구문과 같이 자료 조작이 일어나는 구문들에 대한 EXPLAIN ANALYZE 작업이 필요하다면, 기존 자료 보호를 위해서 다음과 같은 방법으로 사용하길 권한다.

BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;

	QUERY PLAN
1	Hash Join (cost=15.31..24.54 rows=279 width=13)
2	Hash Cond: ((a2.stg_cd)::text = (a1.stg_cd)::text)
3	-> Seq Scan on tb_cmn_stg_cha_corse_info a2 (cost=0.00..8.49 rows=279 width=13)
4	Filter: ((chn_scn_cd)::text = 'BRS'::text)
5	-> Hash (cost=11.25..11.25 rows=325 width=8)
6	-> Seq Scan on tb_cmn_stg a1 (cost=0.00..11.25 rows=325 width=8)