Table of Contents

1. 문서 개요

1.1. 목적

본 문서는 CIS 프로젝트에 적용하는 SQL 문 작성시 표준을 제시하고, SQL 문에 대한 이해를 돕기 위해서 기본적으로 사용되는 SQL 문 및 최적의 성능을 내기 위한 적절한 구문사용 방법을 설명하고자 한다.

1.2. 대상

본 문서는 다음과 같은 역할을 수행하는 대상으로 작성되었다.

- › ↓ ↓ ↓ ↓ ↓↓↓CIS 구축 프로젝트 현대해상 DA › CIS 구축 프로젝트 모델팅/DBA
- › ↓ ↓ ↓ ↓ ↓↓↓Application 개발자

-

1.3. 가정 및 제약 사항

1.4. 관련 문서

본 문서와 관련된 산출물 및 참고 자료는 다음과 같다.

[표 1-1] 관련문서

문서번호	문서명	문서 내용

2. SQL 문 기본지침

2.1. 작성 표준

SQL 문 코딩규칙은 아래와 같다. 동일한 SQL 문은 Parsing 이 불필요하기에 SQL 문을 공유함으로써 데이터베이스내의 SGA 영역에서 메모리 사용 감소와 hard parsing 을 감소함으로써 빠른 수행을 하게 된다. 그러므로 SQL 문을 공유하기 위해 아래의 표준을 따르도록 한다.

- SQL 구문은 모두 대문자로 사용하도록 한다.
- 콤마 후에는 한 칸을 띄운다.
- 계속되는 콤마는 줄 앞에 위치시킨다.
- Table 의 alias 명을 T1,T2,T3,T4 순으로 부여한다.
- 괄호 사용시는 괄호시작점 과 끝점에 공백을 주지 않는다.
- <,> 사용시는 항상 = 과 함께 사용한다.
- 연산자 기준으로 양쪽에 공백을 준다.
- FROM 절은 select-list 절 다음 줄 앞에 위치시킨다.
- WHERE 절은 FROM 절 다음 줄 앞에 위치시킨다.
- GROUP BY 절은 WHERE 절 다음 줄 앞에 위치시킨다.
- ORDER BY 절은 맨 마지막 줄 앞에 위치시킨다.
- Where 절의 and 는 줄 앞에 위치시키며, 항상 두 칸 공백을 준다.
- Subqeury 또는 Inline view 는 항상 괄호로 묶는다.
- SQL 문 안에는 comment 를 사용하지 않는다.
- SELECT * 를 사용하지 않으며, 필요한 컬럼을 명시한다.
- INSERT ~ VALUES 절내에 컬럼을 지정한다.

[기본 SQL syntax]

```
SELECT column_name1, column_name2
                , column_name3
FROM table_name T1, table_name T2
WHERE expr operator
      AND expr operator
GROUP BY column_name
ORDER BY column_name
```

```
INSERT INTO table_name (COL1,COL2)
VALUES(:V_COL1,:V_COL2)

UPDATE table_name
        SET column_name, ….
WHERE expr operator
```

2.2. 작성 가이드

SQL 작성시 성능적인 측면에서 고려하여 작성하여야 한다. SQL 작성 가이드는 아래와 같다.

- 가능하면 항상 Bind 변수를 사용한다.
- Bind 변수 또는 일반 변수를 포함한 절은 where 절의 맨 마지막에 위치시킨다.
- Where 절에 사용되어지는 date/timestamp type 컬럼은 항상 between 또는 "=" 연산자를 사용한다.
- Where 절에 사용되어지는 date/timestamp/number type 컬럼은 like 연산자를 사용할 수 없다.
- Where 절에 부정형 조건을 사용하지 않는다.
- Where 절에 가급적 OR 사용을 하지 않으며, UNION ALL 을 활용한다.
- Where 절에 Index 를 사용할 컬럼에 변형을 가하지 않는다.
- LIKE 연산자 사용시 비교문자 앞에 '%'를 사용하지 않는다.
- FROM 절에 사용되어지는 table 은 driving 순서대로 오른쪽에서 왼쪽으로 나열한다.
- 조인의 연결고리가 되는 컬럼들의 데이터 타입을 같게 한다.
- Result set 이 한 건이라면 where 절 마지막에 rownum=1 을 추가한다.
- UNION 에 대신에 UNION ALL 사용한다.
- DISTINCT 대신에 EXISTS 를 사용한다.
- Select-list 에 필요한 컬럼만 명시한다.("*" 사용하지 않는다.)
- 가능하다면 SQL 호출 수를 줄여야 한다.

2.2.1. 가능하면 항상 Bind 변수를 사용한다.

Literal SQL 로 작성되면 library cache 내에서 항상 다른 SQL 로 인식되어 공유될 수 없으므로 수행 시마다 매번 parsing 을 수행해야 한다. 이와 같은 Literal SQL 이 DB 성능을 저하시키는 주범이 되는 경우가 많으므로 Bind 변수를 사용하여 hard parsing 이 수행되지 않도록 예방하여야 한다.

예제)

수정 전	수정 후
SELECT PKA_KADR_ID FROM PARTY_ADDRESS WHERE PKA_HSEQNO = 1	SELECT PKA_KADR_ID FROM PARTY_ADDRESS WHERE PKA_HSEQNO = :V_SEQ

Bind 변수를 사용할 경우 기존의 execution plan 을 활용하기 때문에 re-parsing 이 발생하지 않는다.

2.2.2. Bind 변수 또는 일반 변수를 포함한 절은 where 절의 맨 마지막에 위치시킨다.

Optimizer 는 뒤에서부터 parsing 을 하므로 가장 먼저 parsing 되어야 한다면 뒤에 위치시켜야 한다.

예제)

수정 후

```
SELECT PTY_ID, PTY_TID
             , PTY_REG_NO
FROM  PARTY T1, PARTY_ADDRESS_RL T2
WHERE T1.PTY_ID = T2.PKA_PTY_ID
      AND T1.PTY_ID = '12345'
```

2.2.3. Where 절에 사용되어지는 date/timestamp type 컬럼은 항상 between 또는 "=" 연산자를 사용한다

조건절에 사용되어지는 컬럼의 datatype 이 date/timstamp 라면 between 또는 "=" 연산자를 사용하여야 한다. 그 외 연산자를 사용할 경우 컬럼의 변형 및 잘못된 데이터를 유발 할 수 있다.

예제)

수정 후	SELECT PTY_ID, PTY_TID , PTY_REG_NO FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID AND T1.PTY_OCCR_ST_TS BETWEEN :V_TS1 AND :V_TS2

2.2.4. Where 절에 사용되어지는 date/timestamp/number type 컬럼은 like 연산자를 사용할 수 없다.

Date/timestamp/number datatype 을 like 연산자와 비교시 내부 변형으로 인한 Index 를 사용할 수 없으므로 가능하다면 between 연산자를 이용하여야 한다.

예제)

수정 후	SELECT PTY_ID, PTY_TID , PTY_REG_NO FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID AND T1.PTY_OCCR_ST_TS LIKE '20070901% => character datatype 으로 내부 변형 발생

2.2.5. Where 절에 부정형 조건을 사용하지 않는다.

Indexed 컬럼인 경우 부정형 조건을 사용할 경우 Index 를 사용할 수 없기 때문에 부정형 조건을 사용하면 안된다.

예제)

수정 전	수정 후
SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID IS NOT NULL	SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID > 0

2.2.6. Where 절에 가급적 OR 사용을 하지 않으며, UNION ALL 을 활용한다.

Indexed 컬럼에 OR 조건을 복잡하게 사용되면 full-table scan 이 발생하여 성능저하를 초래할 수 있으므로 OR 사용을 가급적 제한해서 사용하여 UNION , UNION ALL 또는 DECODE, IN 등을 사용해서 OR 연산자를 없앨 수 있으면 없애도록 한다.

예제)

수정 전	수정 후

SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID = '12345' OR PTY_TID = '54321'	SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID = '12345' UNION ALL SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_TID = '54321'

2.2.7. Where 절에 Index 를 사용할 컬럼에 변형을 가하지 않는다.

Indexed 컬럼에 대하여 변형을 가할 경우에 Index 를 사용할 수 없으므로 변형을 하지 않도록 주의하여야 한다.

예제 1)

수정 전	수정 후
SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE SUBSTR(PTY_ID,1,5) = '12345'	SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID LIKE '12345%'

예제 2)

수정 전	수정 후
SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID||PTY_REG_NO = '123454212'	SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID = '12345' AND PTY_REG_NO = '4212'

2.2.8. LIKE 연산자 사용시 비교문자 앞에 '%'를 사용하지 않는다.

LIKE 연산자를 사용시 주의 사항은 비교문자 앞에 '%'를 사용할 경우 Index 를 활용할 수 없다.

예제 1)

수정 전	수정 후
SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID LIKE '%12345%'	SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID LIKE '12345%'

2.2.9. From 절에 사용되어지는 table 은 driving 순서대로 오른쪽에서 왼쪽으로 나열한다.

오라클은 조인 수행시 오른쪽에서부터 수행하게 되므로 driving table 은 오른쪽에 나열하여야 한다.

예제)

수정 전	수정 후

SELECT PTY_ID, PTY_TID , PTY_REG_NO, COUNT(PTY_ID) FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID ANDT1.PTY_ID = '12345' GROUP BY PTY_ID, PTY_TID, PTY_REG_NO	SELECT PTY_ID, PTY_TID , PTY_REG_NO, COUNT(PTY_ID) FROM PARTY_ADDRESS_RL T1, PARTY T2 WHERE T2.PTY_ID = T1.PKA_PTY_ID AND T2.PTY_ID = '12345' GROUP BY PTY_ID, PTY_TID, PTY_REG_NO

2.2.10. 조인의 연결고리가 되는 컬럼들의 데이터 타입을 같게 한다.

조인이나 문자 비교시 내부 변형이 발생하지 않도록 같은 datatype 을 사용하여야 한다.

예제 1)

수정 후	SELECT PTY_ID, PTY_TID , PTY_REG_NO, COUNT(PTY_ID) FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T1.PKA_PTY_ID => 같은 Datatype 을 사용 AND T1.PTY_ID = '12345' => 변수사용시 컬럼의 타입 주의 GROUP BY PTY_ID, PTY_TID, PTY_REG_NO

예제 2)

수정 전	SELECT PTY_ID, PTY_REG_NO FROM PARTY WHERE PTY_ID = 123456 Execution Plan ---------- 0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=119K Card=1 Bytes=26) 1 0 TABLE ACCESS (FULL) OF 'PARTY' (TABLE) (Cost=119K Card=1 Bytes=26)
수정 후	SELECT PTY_ID, PTY_REG_NO FROM PARTY WHERE PTY_ID = '123456'

	Execution Plan ---------- 0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=1 Bytes=26) 1 0 TABLE ACCESS (BY INDEX ROWID) OF 'PARTY' (TABLE) (Cost=3 Card=1 Bytes=26) 2 1 INDEX (UNIQUE SCAN) OF 'XPTYX0' (INDEX (UNIQUE)) (Cost=2 Card=1)

2.2.11. Result set 이 한 건이라면 where 절 마지막에 rownum=1 을 추가한다.

Select 의 result set 값이 한 건이라면 rownum 을 이용하여 한 건만 scan 하도록 하여 추가적인 scan 을 방지할 수 있다.

예제)

수정전	SELECT PTY_ID, PTY_REG_NO FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID AND T1.PTY_ID = '005eJREIts1ZE8' Execution Plan ---------- 0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 Card=3 Bytes=123) 1 0 NESTED LOOPS (Cost=6 Card=3 Bytes=123) 2 1 TABLE ACCESS (BY INDEX ROWID) OF 'PARTY' (TABLE) (Cost=3 Card=1 Bytes=26) 3 2 INDEX (UNIQUE SCAN) OF 'XPTYX0' (INDEX (UNIQUE)) (Cost=2 Card=1) 4 1 INDEX (RANGE SCAN) OF 'X$PADX1' (INDEX (UNIQUE)) (Cost=3 Card=3 Bytes=45)
수정후	SELECT PTY_ID, PTY_REG_NO FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID AND T1.PTY_ID = '005eJREIts1ZE8' AND ROWNUM = 1 Execution Plan ---------- 0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 Card=1

```
Bytes=41)
   1     0    COUNT (STOPKEY)
   2     1      NESTED LOOPS (Cost=6 Card=1 Bytes=41)
   3     2        TABLE ACCESS (BY INDEX ROWID) OF 'PARTY' (TABLE)
(Cost=3 Card=1 Bytes=26)
   4     3          INDEX (UNIQUE SCAN) OF 'XPTYX0' (INDEX
(UNIQUE)) (Cost=2 Card=1)
   5     2        INDEX (RANGE SCAN) OF 'X$PADX1' (INDEX (UNIQUE))
(Cost=3 Card=1 Bytes=15)
```

2.2.12. UNION 에 대신에 UNION ALL 사용한다.

UNION 연산자는 sort operation 을 수반하여 결과값을 만들기 때문에 성능저하의 원인이 될 수 있다.그러므로 가능하다면 UNION ALL 을 활용하도록 한다.

예제)

수정전

```
SELECT PTY_ID, PTY_REG_NO
FROM PARTY T1, PARTY_ADDRESS_RL T2
WHERE T1.PTY_ID = T2.PKA_PTY_ID
     AND T1.PTY_ID LIKE '005eJREIts1ZE%'
UNION
SELECT PTY_ID, PTY_REG_NO
FROM PARTY T1, PARTY_ADDRESS_RL T2
WHERE T1.PTY_ID = T2.PKA_PTY_ID
     AND T1.PTY_ID LIKE '005eJREI%'

Execution Plan
----------------------------------------------------------
---------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=14 Card=6
Bytes=246)
   1     0    SORT (UNIQUE) (Cost=14 Card=6 Bytes=246)
   2     1      UNION-ALL
   3     2        NESTED LOOPS (Cost=6 Card=3 Bytes=123)
   4     3          TABLE ACCESS (BY INDEX ROWID) OF 'PARTY'
(TABLE) (Cost=4 Card=1 Bytes=26)
   5     4            INDEX (RANGE SCAN) OF 'XPTYX0' (INDEX
(UNIQUE)) (Cost=3 Card=1)
   6     3          INDEX (RANGE SCAN) OF 'X$PADX1' (INDEX
(UNIQUE)) (Cost=3 Card=3 Bytes=45)
   7     2        NESTED LOOPS (Cost=6 Card=3 Bytes=123)
   8     7          TABLE ACCESS (BY INDEX ROWID) OF 'PARTY'
(TABLE) (Cost=4 Card=1 Bytes=26)
   9     8            INDEX (RANGE SCAN) OF 'XPTYX0' (INDEX
(UNIQUE)) (Cost=3 Card=1)
```

	10 7 INDEX (RANGE SCAN) OF 'X$PADX1' (INDEX (UNIQUE)) (Cost=3 Card=3 Bytes=45)
수정후	SELECT PTY_ID, PTY_REG_NO FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID AND T1.PTY_ID LIKE '005eJREIts1ZE%' UNION ALL SELECT PTY_ID, PTY_REG_NO FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID AND T1.PTY_ID LIKE '005eJREI%' Execution Plan ---------- 0 SELECT STATEMENT Optimizer=ALL_ROWS (Cost=12 Card=6 Bytes=246) 1 0 UNION-ALL 2 1 NESTED LOOPS (Cost=6 Card=3 Bytes=123) 3 2 TABLE ACCESS (BY INDEX ROWID) OF 'PARTY' (TABLE) (Cost=4 Card=1 Bytes=26) 4 3 INDEX (RANGE SCAN) OF 'XPTYX0' (INDEX (UNIQUE)) (Cost=3 Card=1)

```
    5     2         INDEX (RANGE SCAN) OF 'X$PADX1' (INDEX (UNIQUE))
(Cost=3 Card=3 Bytes=45)
    6     1       NESTED LOOPS (Cost=6 Card=3 Bytes=123)
    7     6         TABLE ACCESS (BY INDEX ROWID) OF 'PARTY' (TABLE)
(Cost=4 Card=1 Bytes=26)
    8     7           INDEX (RANGE SCAN) OF 'XPTYX0' (INDEX (UNIQUE))
(Cost=3 Card=1)
    9     6         INDEX (RANGE SCAN) OF 'X$PADX1' (INDEX (UNIQUE))
(Cost=3 Card=3 Bytes=45)
```

2.2.13. DISTINCT 대신에 EXISTS 를 사용한다.

DISTINCT 는 항상 sort operation 을(전체범위 처리) 수반하므로 성능저하를 가져올 수 있으므로 EXISTS 를 사용하여 부분범위 처리가 가능하도록 하거나, 처리 결과값이 항상 unique 한 경우 DISTINCT 키워드를 사용하지 않도록 한다.

예제)

수정 전	수정 후
SELECT DISTINCT PTY_ID FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID	SELECT PTY_ID FROM PARTY T1 WHERE EXISTS(SELECT 'X' FROM PARTY_ADDRESS_RL WHERE PKA_PTY_ID = T1.PTY_ID)

2.2.14. SELECT * 를 사용하지 않으며, 필요한 컬럼을 명시한다.

"*" 사용할 경우 해당 table 의 모든 컬럼을 fetch 하여 핸들링하기 때문에 메모리 낭비를 초래할 수 있다.

예제)

수정 전	수정 후
SELECT * FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID	SELECT T1.PTY_ID, T2.PKA_PTY_ID FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID

SELECT COUNT(*) => SELECT COUNT(Primary key 또는 non-null Index key) 전환하여야 한다.

2.2.15. 가능하다면 SQL 호출 수를 줄여야 한다.

가능하다면 SQL 호출 수를 최소화 시켜야 한다. SQL 호출 수 만큼 parsing 횟수 및 자원 소비가 증가하기 때문에 SQL 호출 수를 아래의 방법을 활용하여 최소화하여야 한다.

예제 1) SQL 통합

다음과 같이 2 개의 SQL 로 처리되는 문장을 하나의 SQL 문으로 통합하여 SQL CALL 횟수를 줄일 수 있다.

수정 전	수정 후
SELECT SYSDATE FROM DUAL; SELECT PTY_ID, PTY_REG_NO FROM PARTY WHERE PTY_ID = ‘12345’;	SELECT PTY_ID, PTY_REG_NO, SYSDATE FROM PARTY WHERE PTY_ID = ‘12345’

예제 2) Loop Query 제거

다음과 같이 Loop 내에서 반복적으로 실행되는 SQL 문은 Loop 횟수 만큼 SQL 호출이 발생되어 심각한 성능 저하를 초래할 수 있다. 따라서 Loop 내에서 SQL 문이 호출되지 않도록 하여야 한다.

수정 전	수정 후
DECLARE CURSOR C1 FOR SELECT PTY_ID FROM PARTY WHERE PTY_ID LIKE ‘123%’ OPEN CURSOR C1; FETCH C1 INTO :V_ID LOOP UNTIL NOT FOUND SELECT PKA_ID FROM PARTY_ADDRESS_RL WHERE PKA_PTY_ID = V_ID; END LOOP;	SELECT PTY_ID, PKA_ID FROM PARTY_ADDRESS_RL T1, PARTY T2 WHERE T1.PKA_PTY_ID = T2.PTY_ID AND T1.PTY_ID LIKE ‘123%’

예제 3)DECODE 를 이용한 SQL 문 통합

다음과 같이 동일한 테이블을 사용하면서 WHERE 조건이 비슷한 유형으로 전개되는 SQL 문은 DECODE 함수를 사용하여 하나로 통합될 수 있다.

수정 전	SELECT PKA_PTY_ID, MAX(PKA_LUPD_CNT) FROM PARTY_ADDRESS_RL WHERE PKA_PTY_ID = ‘005eJREIts1ZE8’; SELECT PKA_PTY_ID, COUNT(PKA_PTY_ID) FROM PARTY_ADDRESS_RL WHERE PKA_LUPD_CNT = 1 AND PKA_PTY_ID = ‘005eJREIts1ZE8’;
수정 후	SELECT MAX(PKA_LUPD_CNT)

```
        , DECODE(PKA_LUPD_CNT,1,COUNT(PTY_ID),0)
FROM PARTY_ADDRESS_RL
WHERE PKA_PTY_ID = '005eJREIts1ZE8'
GROUP BY PKA_LUPD_CNT
```

2.3. CIS 추가 제약사항

- IS NULL 또는 IS NOT NULL 을 사용하지 않는다.
- NVL 함수를 사용하지 않는다.
- SQL 문 안에는 Hint 를 사용하지 않는다. Hint 를 사용해야 하는 경우는 DBA 검증 후 사용한다.
- Tuning 시점 이전까지는 인덱스는 기본 인덱스(Primary key, Foreign key, Business Key)만 존재한다.
- 작성된 SQL 문에 대한 실행계획을 항상 확인하여 검증하도록 한다.
- Subquery 또는 Inline view 는 1 level 까지 허용한다.
- Function 은 2 level 까지 허용하며, 동일한 function 을 연속적으로 사용할 수 없다.
- DECODE 연산자는 1 level 만 허용한다.

2.3.1. IS NULL 또는 IS NOT NULL , NVL 함수를 사용하지 않는다.

물리 DB 설계시 모든 컬럼에 대한 default value 를 가지므로 NULL check 가 필요하지 않다.

2.3.2. SQL 문 안에는 Hint 를 사용하지 않는다. Hint 를 사용해야 하는 경우는 DBA 검증 후 사용한다.

힌트를 사용하지 않으며, 인덱스 구성이 완료되고 Tuning 시점에 DBA 와 협의하여 힌트를 사용하도록 한다.

2.3.3. Tuning 시점 이전까지는 인덱스는 기본 인덱스(Primary key, Foreign key, Business key)만 존재한다.

Tuning 시점 이전까지는 Primary key index 와 Foreign key Index, Business key Index 를 생성할 것이다.
차후 Tuning 시점에서 SQL 문장 대비 Index 를 설계할 것이다.

2.3.4. 작성된 SQL 문에 대한 실행계획을 항상 확인하여 검증하도록 한다.

ORACLE 10G 부터 Optimizer 는 CBO 방식으로 운영되기 때문에, SQL 문장이 원하는 방식대로 Execution plan 이 풀렸는지 확인을 하여야 한다.

2.3.5. Subquery 또는 Inline view 는 1 level 까지 허용한다.

Subquery 또는 Inline view 는 1 level 까지 사용 가능하며, subquery 내 또 subquery 는 사용할 수 없다.
Subquery 에 대한 자세한 내용은 4 장에서 언급하고자 한다.

2.3.6. Function 은 2 level 까지 허용하며, 동일한 function 을 연속적으로 사용할 수 없다.

오라클에서 자주 사용되어지는 function 은 1 level 까지 사용하여야 지원한다. 하지만 동일한 function 인 경우는 연속적으로 사용할 수 없으며, 필요한 경우에만 function 을 2 level 까지 사용하도록 한다.

- Number Functions

  : ABS, CEIL, COS, EXP, FLOOR, MOD, POWER, ROUND, SIGN, TRUNC

- Character Functions
  : CONCAT, LOWER, UPPER, LTRIM, RTRIM, TRIM, REPLACE, SUBSTR
- Date Functions
  : ADD_MONTHS, CURRENT_DATE, CURRENT_TIMESTAMP, LAST_DAY, MONTHS_BETWEEN, SYSDATE, SYSTIMESTAMP
- Conversion Functions
  : TO_CHAR, TO_DATE, TO_TIMESTAMP, TO_NUMBER
- Aggregate Functions
  : AVG, COUNT, MAX, MIN, RANK, SUM
- 그 외 Functions
  : DECODE, CASE, DISTINCT

예제)

수정 전	수정 후
1 level 인 경우 : PTY_ID 별 개수 구하기 SELECT PTY_ID, COUNT(PTY_ID) FROM PARTY GROUP BY PTY_ID	2 level 인 경우 : PTY_ID 별 개수에서 가장 큰 값 구하기 SELECT PTY_ID, MAX(COUNT(PTY_ID)) FROM PARTY GROUP BY PTY_ID

2.3.7. DECODE 연산자는 1 level 만 허용한다.

하나의 DECODE 내에 여러 개의 비교를 연속해서 사용하는 것 보다 DECODE 내에 DECODE 가 반복되는 형식이 훨씬 overhead 가 크다. 따라서 DECODE level 이 증가하는 경우 아래와 같이 컬럼을 경합하거나 함수 등을 사용해서 단순한 DECODE 로 만들어주어야 한다.

예제)

수정 전	수정 후
SELECT SUM(DECODE(PTY_LUPD_CNT,1, DECODE(PTY_TID ,1,0)),0)) FROM PARTY	SELECT SUM(DECODE(PTY_LUPD _CNT|| PTY_TID,11,1,0)) FROM PARTY

3. JOIN

Oracle 의 JOIN Method 는 Nested Loop Join, Sort Merge Join, Hash Join 의 3 가지가 있다. 이 3 가지 JOIN Method 장단점을 이해하고, 사용시 효과적으로 사용할 수 있도록 한다.

3.1. Nested Loop Join

- ・・・ ··Nested-Loop Join 는 순차적인 처리로 fetch 의 운반단위(Array Size)마다 결과 row 를 리턴 받을 수 있다.
- ・・・ ··Join Method 방법 중 첫 번째 row 를 받는 시간이 가장 빠르다.
  - 별도의 결과 대기 영역이 필요 없다.
  - FIRST_ROWS 에 최적이다.
- ・・・ ··Index 의 효율이 좋다면 가장 최적을 성능을 발휘하는 Join Method 이다.

- Index 를 적절히 설정하면 가장 빠른 성능을 보장한다.
- Index 를 잘 못 설계한 경우 최악의 성능이 나올 수 있다.

● ・・・ ··Nested-Loop Join 는 주로 Index 위주의 single block I/O 의 random I/O 위주이므로 OLTP 에서 적은 데이터 범위 처리에 주로 사용된다. 즉, 전체의 15% 이상의 경우는 Full Table Scan 을 이용한 Sort Merge 또는 Hash Join 을 이용한다.

● ・・・ ··추가 적인 메모리 비용이 필요하지 않다.

● ・・・ ··Join 순서
- Driving table 의 선택은 범위를 가장 줄여줄 수 있는 table 부터 시작한다.
- inner table 의 access path 에 매우 민감하다.
- from 절의 테이블 순서 조정도 중요한 point 가 될 수 있다.
- 불필요한 outer join 을 사용하지 않도록 한다.

3.2. Sort Merge Join

● ・・・ ··전체범위 처리가 기본이다.

● ・・・ ··전체 데이터를 리턴 받는 시간이 Nested-Loop Join 보다 빠르다.

● ・・・ ··추가적인 sort 메모리(sort_area_size) 비용이 필요하다. 메모리 공간이 부족하다면 temp tablespace 을 사용하게 된다.

● ・・・ ··Sort 메모리에 위치하는 대상은 Join key 뿐만 아니라 select list 도 포함하므로 불필요한 select list 는 제거해야 한다.

● ・・・ ··Sort 를 요구하지 않는 경우는 Hash Join 이 유리하다.

① PARTY_IDX Index 를 통해 PARTY Table 에서 만족하는 row 를 추출한다.
② 추출한 데이터를 sort 를 해둔다.
③ PARTY_RL table 에서 조건을 만족하는 row 를 추출한다.
④ 추출한 데이터를 sort 를 해둔다.
⑤ Sort 한 데이터를 scan 하면서 조건에 만족하는 row 를 추출해서 운반단위로 fetch 한다.

3.3. Hash Join

● ・・・ ··CBO 에서만 가능하며, CPU Power 에 의존적이다.(hash 연산)

● ・・・ ··Hash Join 은 equal(=) Join 만 가능하다.

● ・・・ ··Hash Join 은 두 개의 Join Table 중 small rowset 을 가지고 hash_ares_size 에 지정된 메모리 내에서 hash table 을 만든다.

● ・・・ ··Hash table 을 만든 이후부터는 Nested-Loop Join 의 장점인 순차적인 처리 형태로 수행된다.

● ・・・ ··Hash Join 은 Nested-Loop Join 과 Sort Merge Join 의 장점을 가지고 있다.

[표 3.1 Join Method 비교]

Join Method	Nested Loop	Hash	Sort Merge
처리 방식	순차적(완전 부분범위)	반 부분범위	동시적(전체범위처리)
Access 방식	Random Access	Hash Function	Scan 방식
연결 고리	절대 영향	영향 없음	영향 없음
Join 방향	영향 큼	영향 있음 (Hash Table 구성)	영향 없음
사용 resource	BUFFER CACHE	PGA	PGA
처리량	좁은 범위에 유리	넓은 범위에 유리	넓은 범위에 유리
주요 Check 요소	연결 고리 상태 및 처리량	Hash_area_size	Sort_area_size
		Hash Table size	각 Table 의 Sort 량

			(Temp 사용량)

3.4. Outer Join

- ・・・ ‥Join 조건에 만족되지 않더라도 Join 요건이 성립되는 방법이다.
- ・・・ ‥Join 순서는 항상 고정적이다.
- ・・・ ‥Join 순서가 미리 정해져 있으므로 Join 순서를 이용한 Tuning 은 불가능하며, 가능하다면 Outer Join 을 사용하지 않도록 한다.
- ・・・ ‥Outer Join 대상 Table 에 대한 모든 조건에 (+) 기호를 사용하여야 하며, 조건 하나라도 누락된 경우는 Outer Join 이 성립되지 않는다.
- ・・・ ‥(+) 기호를 이용하여 IN, OR 의 연산자를 이용할 수 없다. IN, OR 연산자를 사용하기 위해서는 Inline View 를 통해서 해결이 가능하다.
- ・・・ ‥(+) 기호를 이용하여 Subquery 와 비교할 수 없다.

- IN, OR 연산자를 사용하기 위해서는 Inline View 를 통해서 해결 예제

수정 전	SELECT * FROM TAB1 T1, TAB2 T2 WHERE T1.KEY = T2.KEY(+) AND T1.FLD1 > 'AAA' AND T2.COL1(+) IN ('10', '20', '30') ; -> 에러 발생(ORA-01719)
수정 후	SELECT * FROM TAB1 T1, (SELECT KEY FROM TAB2 WHERE COL1 IN ('10', '20', '30')) T2 WHERE T1.KEY = T2 .KEY(+) AND T1.FLD1 > 'AAA'

4. Subquery 활용

Subquery 는 select-list 와 where 절, having 절, from 절에 사용이 가능하며, 연산자 >,=,IN 등 사용이 가능하다.

[기본 Subquery Syntax]

```
SELECT select_list
FROM table_name
WHERE expr operator
                    (SELECT
select_list
                    FROM
table_name)
```

4.1. Subquery 작성 가이드

- ・・ ‥‥Subquery 는 조건절 우측에 위치 시킨다.
- ・・ ‥‥Subquery 는 1 level 만 허용한다.
- ・・ ‥‥Where 절, having 절의 subquery 는 NULL value 가 나오지 않아야 한다.
- ・・ ‥‥Subquery 내에 Top-n analysis 를 제외한 구문에서는 ORDER BY 를 사용할 수 없다.

- ・・・Subquery 의 result set 에 따라 연산자(single-row, multiple-row operator)를 사용하여야 한다.
  - single-row subquery 인 경우 사용 가능한 operator.
    : =, >, >=, <, <=, <> 등
    - multiple-row subquery 인 경우 사용 가능한 operator
      : IN, ANY, ALL 등

4.1.1. Select-list 의 Subquery 사용방법

[Select-list 의 Subquery Syntax]

```
SELECT select_list
        , (SELECT select_list
           FROM table_name
      WHERE expr operator)
FROM table_name
WHERE expr operator
```

- ・・・Select-list 의 subquery 는 항상 single-row 만 리턴이 되어야한다.
- ・・・Mainquery 와 subquery 간 Join 이 가능하다.

4.1.2. FROM 절의 subquery 사용방법(Inline View)

[Select-list 의 Subquery Syntax]

```
SELECT select_list
FROM (SELECT select _list
          FROM table_name
          WHERE expr operator
          GROUP BY column_name)
WHERE expr operator
```

- ・・・FROM 절의 subquery 는 GROUP BY 절을 통해 데이터를 최소하여 사용할 경우에 사용한다.
- ・・・Top-n analysis 구문을 제외한 다른 구문에서는 ORDER BY 를 사용하지 않는다.

4.1.3. WHERE 절, HAVING 절의 subquery 사용방법

[WHERE 절, HAVING 절의 Subquery Syntax]

```
SELECT select_list
FROM table_name
WHERE expr operator
                                    (SELECT
select_list
                                      FROM
table_name)
HAVING expr operator
                                    (SELECT
select_list
                                      FROM
table_name)
```

- · · ···Subquery result value 가 NULL 이 나오지 않아야 한다.
- · · ···result set 에 따라 연산자(single-row, multiple-row operator)를 사용하여야 한다.

5. SQL 작성시 주의 사항

5.1. 부분 범위 처리

부분 범위 처리란 Fetch 된 데이터 순서로 운반단위(ARRAY SIZE)에 도달하면 멈추게 하는 처리방법을 말하며, 전체 범위 처리는 전체 데이터를 가공하여 운반단위만큼 추출하는 처리 방법을 말한다.

전체 범위 처리	부분 범위 처리
Full Table Scan 후 가공하여 운반단위에 추출	조건을 만족하는 데이터수가 운반단위에 도달하면 멈춤

SUM,COUNT,MAX,MIN,AVG 등의 GROUP 함수를 사용하거나 GROUP BY, ORDER BY, UNION, MINUS 등의 OPERATOR 를 사용하면 부분 범위 처리를 할 수 없다. 꼭, 전체 범위 처리를 해야 하는 경우를 제외하고 대부분 부분 범위 처리방법으로 처리할 수 있으며 전체 범위 처리를 하는 경우보다 성능향상을 가져 올 수 있다.

부분 범위 처리는 조건을 만족하는 데이터를 운반단위로 데이터를 가져오기 때문에 데이터가 많아도 성능에 지장을 주지 않는다.

아래와 같은 경우에 부분 범위 처리를 함으로서 성능 향상을 가져올 수 있다.

- · ··SORT Operation 을 Index 로 대체함.
- · ··TABLE 은 ACCESS 하지않고 INDEX 만 사용하도록 유도
- · ··MAX 처리시 INDEX_DESC hint 를 활용함.
- · ··EXISTS 의 활용
- · ··ROWNUM 의 활용

5.1.1. SORT Operation 부분 범위 처리

ORDER BY, GROUP BY 사용할 경우 SORT 작업으로 인한 전체 범위 처리를 하게 된다. 하지만 아래와 같이 INDEX_DESC hint 를 사용하면 전체 범위 처리에서 부분 범위 처리로 변경 가능하다.

전체 범위 처리	부분 범위 처리

SELECT PTY_ID, PTY_REG_NO FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID AND T1.PTY_ID LIKE '005%' ORDER BY PTY_ID	SELECT /*+ INDEX_DESC(A XPTYX0) */ PTY_ID, PTY_REG_NO FROM PARTY T1, PARTY_ADDRESS_RL T2 WHERE T1.PTY_ID = T2.PKA_PTY_ID AND T1.PTY_ID LIKE '005%' ORDER BY PTY_ID

5.1.2. Index 를 이용한 부분 범위 처리

INDEX 를 이용하여 전체 범위 처리에서 부분 범위 처리로 변경 가능하다.

전체 범위 처리(INDEX : PTY_REG_NO)	부분 범위 처리(INDEX : PTY_REG_NO+PTY_ID)
SELECT PTY_REG_NO, COUNT(PTY_ID) FROM PARTY WHERE PTY_REG_NO '5%' GROUP BY PTY_REG_NO	SELECT PTY_REG_NO, COUNT(PTY_ID) FROM PARTY WHERE PTY_REG_NO '5%' GROUP BY PTY_REG_NO

5.1.3. Group function 시 부분 범위 처리

MAX,MIN 등 Group function 을 사용하여 query 작성할 경우 일반적으로 전체 범위 처리로 수행하게 된다. 하지만 INDEX_DESC/INDEX hint 를 활용하면 전체 범위 처리에서 부분 범위 처리로 변경 가능하다.

전체 범위 처리(INDEX : PTY_ID)	부분 범위 처리(INDEX : PTY_ID)
SELECT MAX(PT_LUPD_CNT)+1 FROM PARTY WHERE PTY_ID = '123454'	SELECT /*+ INDEX_DESC(A XPTYX0) */ MAX(PT_LUPD_CNT)+1 FROM PARTY WHERE PTY_ID = '123454'

5.1.4. EXISTS 를 이용한 부분 범위 처리

EXISTS 를 활용부분은 2.2.12 사항을 참고.

5.1.5. ROWNUM 를 이용한 부분 범위 처리

ROWNUM 를 이용한 부분 범위 처리는 로직으로 처리되는 SQL 문장에 대해서 CHECK 조건으로 활용할 경우에 활용이 가능하다.

전체 범위 처리 : 로직상 CHECK 조건이 필요한	부분 범위 처리 : 로직상 CHECK 조건이 필요한

경우	경우
SELECT COUNT(PTY_ID) INTO :CNT FROM PARTY WHERE PTY_ID = '123456' … IF CNT > 0 …	SELECT 1 INTO :CNT FROM PARTY WHERE PTY_ID = '123456' AND ROWNUM = 1 … IF CNT > 0 …

5.2. Index 처리가 불가능한 경우

SQL 문 최적화를 위해 대부분 Index 처리를 하도록 하고 있다. 하지만 해당 Table 에 Index 가 있더라도 아래와 같은 경우에는 Index 를 활용하지 못한다. 아래의 경우를 예제를 통하여 확실히 숙지하여 Index 를 사용할 수 있도록 SQL 문을 작성하여야 한다.

- · ··Index 컬럼의 변형(외부/내부)
- · ··Index 컬럼의 조건절을 부정형으로 사용할 경우 (!= , NOT IN)
- · ··IS (NOT) NULL 로 비교하는 경우.(TO-BE 물리 모델은 모두 NOT NULL 이므로 NULL 대한 부분은 언급을 생략함.)
- · ··LIKE 연산자로 인한 컬럼의 내부 변형이 발생하는 경우.(datatype 변형/연산자 사용주의)
- · ··Having 절 사용하여 비교하는 경우.
- · ··IN 연산자를 사용하는 경우

5.2.1. Index 컬럼의 변형으로 인한 Index 를 사용할 수 없는 경우

Index 컬럼에 대한 내부 변형이나 외부 변형으로 인하여 Index 를 활용하지 못하는 경우를 아래의 예시로 확인해 보자.

Index 를 사용하지 못하는 경우(내부변형)	Index 를 사용하는 경우
Index 컬럼 : PTY_ID(unique Index) 컬럼의 datatype : CHAR SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID = 123451234 => 변수값이 상수로 들로 왔기 때문에 PTY_ID 는 상수로 datatype conversion 이 발생함.	Index 컬럼 : PTY_ID(unique Index) 컬럼의 datatype : CHAR SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID = '123451234' => 해당 컬럼의 datatype 과 항상 동일하게 변수를 선언을 하여 사용하여야 함.

Index 를 사용하지 못하는 경우(외부변형)	Index 를 사용하는 경우
Index 컬럼 : PTY_ID(unique Index) 컬럼의 datatype : CHAR SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE SUBSTR(PTY_ID,1,5) = '51234' => 해당 컬럼에 substr 이라는 함수로 인한 변형이 발생하여 Index 를 사용할 수 없음.	Index 컬럼 : PTY_ID(unique Index) 컬럼의 datatype : CHAR SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_ID LIKE '51234%' => 해당 컬럼을 LIKE 연산자를 이용하여 컬럼이 원상태로 유지할 수 있도록 SQL 수정함.

5.2.2. Index 컬럼의 조건절을 부정형으로 사용할 경우.(NOT,!=,<>)

Index 컬럼의 조건절을 부정형으로 비교할 경우 해당 컬럼의 Index 를 사용할 수 없는 경우를 아래의 예시로 확인해 보자.

Index 를 사용하지 못하는 경우(부정형)	Index 를 사용하는 경우
Index 컬럼 : PTY_LUPD_CNT 컬럼의 datatype : NUMBER SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_LUPD_CNT != 1 => 해당 컬럼과 부정형으로 비교할 경우 Index 를 활용할 수 없음.	Index 컬럼 : PTY_LUPD_CNT 컬럼의 datatype : NUMBER SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_LUPD_CNT > 1 => 부정형 조건절을 적절히 변경하여 Index 를 활용할 수 있도록 수정함.

5.2.3. LIKE 연산자로 인한 컬럼의 내부 변형이 발생하는 경우.

LIKE 연산자는 연산의 대상이 되는 컬럼이 character type 이 아닌 경우는 해당 컬럼을 character datatype 으로 변형이 발생하기 때문에 해당 Index 를 사용할 수 없다. 해당 경우를 아래의 예시로 확인해 보자.

Index 를 사용하지 못하는 경우(내부변형)	Index 를 사용하는 경우
Index 컬럼 : PTY_OCCR_ST_TS+PTY_ID 컬럼의 datatype : DATE,CHAR	Index 컬럼 : PTY_OCCR_ST_TS+PTY_ID 컬럼의 datatype : DATE,CHAR

SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_OCCR_ST_TS LIKE '20070903%' AND PTY_ID = '123456' => LIKE 연산자로 인한 해당 컬럼의 NUMBER datatype 이 CHAR datatype 으로 변형이 발생하여 Index 를 사용할 수 없음.	SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_OCCR_ST_TS BETWEEN TO_DATE('20070903','YYYYMMDD') AND TO_DATE('20070903','YYYYMMDD')+0.99999 AND PTY_ID = '123456' => 해당 컬럼에 대하여 변형이 발생하지 않도록 SQL 문장을 수정함.

5.2.4. HAVING 절을 사용하여 비교하는 경우

데이터 필터링을 위해서 having 절 보다는 where 을 사용하는 것이 유리하다. Index 컬럼의 경우 having 절에 사용을 하게 되면 group by 에 의해 이미 데이터가 grouping 된 결과로 having 절에 쓰이기 때문에 Index 로 활용할 수 없다.

Index 를 사용하지 못하는 경우	Index 를 사용하는 경우
Index 컬럼 : PTY_TID 컬럼의 datatype : NUMBER SELECT PTY_TID, COUNT(PTY_TID) FROM PARTY WHERE PTY_LUPD_CNT = 1 GROUP BY PTY_TID HAVING PTY_TID < 10 => 해당 컬럼이 grouping 이 이루어진 후 having 에 사용되고 있기 때문에 Index 를 활용할 수 없음.	Index 컬럼 : PTY_TID 컬럼의 datatype : NUMBER SELECT PTY_TID, COUNT(PTY_TID) FROM PARTY WHERE PTY_LUPD_CNT = 1 AND PTY_TID < 10 GROUP BY PTY_TID => having 을 where 조건절로 변경할 경우 grouping 전에 Index 를 활용하기 때문에 Index 활용이 가능함.

5.2.5. IN 연산자를 사용할 경우

IN 연산자를 사용할 경우 Index 활용을 하나, list 의 개수가 많이 들어올 경우 Index 를 활용하지 못하는 경우도 있다.

Index 를 사용하지 못하는 경우	Index 를 사용하는 경우
Index 컬럼 : PTY_TID 컬럼의 datatype : NUMBER	Index 컬럼 : PTY_TID 컬럼의 datatype : NUMBER

SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_TID IN (1,2,3,4,5) => ORACLE Optimizer 가 판단하여 list 개수가 많다면 Index 를 활용하지 않음	SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_TID = 1 UNION ALL SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_TID = 2 UNION ALL SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_TID = 3 UNION ALL SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_TID = 4 UNION ALL SELECT PTY_ID, PTY_TID, PTY_REG_NO FROM PARTY WHERE PTY_TID = 5 => Index 를 활용할 수 있도록 list 마다 개별적으로 수행함. * 분명 IN 연산자 사용시 ORACLE Optimizer 가 Full table scan 을 할지 Index scan 을 할지 판단을 하게 된다. 그러므로 IN 연산자를 무조건 UNION ALL 로 개별 수행할 필요는 없다. Plan 정보를 확인한 후 가장 저렴한 COST 쪽으로 SQL 문장을 작성하여야 한다.