2. Oracle

Created	@2021년 9월 30일 오후 6:10
Last Edited Time	@2021년 10월 4일 오후 7:53
Type	DataBase Oracle
Created By	
Last Edited By	

1. 개요

- 본 문서는 SQL문의 효율적인 수행을 위해 SQL문 코딩 기준을 정리하고 개발자가 적용할 수 있도록 한 가이드이다.
- SQL문 작성 시 코딩 표준 및 가이드 준수를 통해 가독성과 재사용율을 향상시켜 유지보수가 용이하도록 하며, 동시에 보다 효율적인 SQL 작성을 통해 DB 성능 저하를 예방하는 것에 목적이 있다.

2. SQL 작성 기본 지침

2.1 SQL 작성 표준

SQL문 코딩 규칙은 아래와 같다. 동일한 SQL문은 re-parsing을 하지 않고 SQL 문을 공유함으로써 데이터베이스 서버 안에서 메모리 사용 감소와 같은 빠른 수행을 하게 된다.

- `SQL`문은 모두 `대문자`로 통일한다.
- `SQL`문의 정렬은 첫 번째 `keyword` 오른 쪽 끝에 세로 줄 `정렬`을 기본으로 한다.
- 테이블 앞에 반드시 `OWNER`를 명시하도록 한다.
- TAB은 사용하지 않는다. 반드시 `space`(공백)을 사용한다.
- 단어 간 공백, 연산자 양쪽 공백은 한 칸을 준수한다.
- 괄호 사용 시는 괄호 안 인접에는 공백을 주지 않는다.
- 함수명과 괄호 사이에는 공백을 주지 않는다.

- SQL 구문에 공백 라인이 없도록 작성한다.
- 테이블과 테이블, 컬럼과 컬럼은 줄 바꿈하여 한 줄에 하나 씩 다른 라인에 나열하도록 한다.
  - 콤마는 공백을 사이에 주고 컬럼(테이블) 앞에 위치시킨다.
  - 컬럼의 개수가 과도하게 많은 경우, 한 라인에 컬럼 목록 나열을 허용하되 가독성을 위해 다음 가이드라인을 따르도록 한다.
    - 한 라인에 5개 이상의 컬럼을 나열하지 않는다.
    - 한 라인에 콤마가 나열될 시 콤마 앞에는 공백이 없고 콤마 다음에 공백을 준다.
    - 컬럼 `ALIAS`를 정의하거나 수식, 함수 등을 사용 시, 단독 라인에 하나씩 위치시킨다.
- 테이블에 반드시 `ALIAS`를 지정하도록 한다.
- SQL상의 모든 컬럼 앞에는 반드시 테이블 `ALIAS`를 누락 없이 기입한다.
- 특정 구문에 주석 필요 시 해당 LINE 오른쪽에 `/* */`으로 작성한다.
- SQL 작성 시 `Oracle Native SQL` 방식을 표준으로 하며, `ANSI SQL`은 사용하지 않도록 한다.
- `Oracle Hint`를 임의로 추가, 변경하지 않도록 한다. 튜닝 관점에서 반드시 `DBA`와 협의 후 적용한다.
- 호출 프로그램 추적을 위해 `SQL` 식별자를 정해진 포맷에 맞춰 반드시 기입한다.
- `SQL`문의 시작 시 `select` 키워드 옆에 `/*+ */`을 이용하여 작성자와 쿼리를 호출하는 메소드 정보를 기입한다.
  - `SQL` 종료자 명시 필요 시 세미콜론`;` 을 사용하여 SQL 문장 끝에 공백없이 기입한다.

작성예시

```
SELECT /*+ 홍길동::com.team.project.member.model.dao.MemberDaoImpl.selectMember() */
       A.회원이름
     , A.회원아이디
     , A.회원전화번호
     , (CASE WHEN A.회원가입일 > :기준년월일 THEN '1' ELSE '0' END) 장기회원
     , SUBSTR(A.주민등록번호, 1, 6) 생년월일
     , B.게시물번호
     , B.게시물제목
  FROM MYBATIS.MEMBER A
     , MYBATIS.BOARD B
 WHERE A.회원번호 = B.작성자번호
```

```
   AND A.회원탈퇴여부 = 'N'
   AND B.게시물삭제여부 = 'N'
   AND NOT EXIST (SELECT 1
                    FROM MYBATIS.MEMBER C
                   WHERE C.회원번호 = A.추천인번호
                     AND C.회원탈퇴여부 = 'N'
                  )
ORDER BY A.회원아이디
       , A.회원명
       , B.게시물번호;
```

2.2 세부 작성 가이드

2.2.1. FROM Clause

- `JOIN` 순서대로 `table`을 나열한다.
  - `ERD`를 기준으로 `JOIN` 조건을 판단하여 가장 먼저 `LEADING` 되는 테이블을 먼저 기술하고(성능적인 고려가 힘들면 논리적인 접근 측면에서) 이후 순차적으로 관련 테이블을 기술하여 `JOIN` 순서의 의도를 쉽게 이해할 수 있도록 한다.
  - 단순 정보를 가져오는 목적의 코드성 테이블들은 뒤로 위치시킨다.
- 기본적으로 `JOIN`은 동일 업무 내에서만 가능하며 타 업무 `JOIN`이 필요한 경우 타 업무 `OWNER`의 `승인`을 득해야 `권한`이 부여된다.
- `ALIAS` 명칭은 영문 대문자 1자리로 통일한다.
  - 알파벳 순으로 순차 명명을 원칙으로 한다. (A, B, C, ...)
    - `FROM`절의 조인 대상 테이블이 많거나 복잡한 `INLINE VIEW`의 사용으로 가독성이 떨어져 조인 대상 테이블간의 계층 확인이 어려운 경우 `ALIAS`를 통해 `DEPTH` 확인이 가능하도록 한다.
    - ex) A01, A02, A11, A12
  - 테이블이 1개인 경우에도 반드시 `ALIAS`를 지정하도록 한다. (추후 튜닝 필요 시 `HINT` 작성 대비 목적)
  - 한 `SQL`문에서 같은 `ALIAS` 명칭을 사용하지 않도록 주의한다. (서브쿼리, 인라인뷰 내부 `ALIAS`가 `MAIN QUERY`와 중복되지 않게 선언한다.)
- 테이블 소유자 (OWNER)를 반드시 작성한다. `(SYNONYM` 미허용)
- `INLINE VIEW`가 있는 경우에는 다음과 같이 `(`를 맞추어 기술한다.

`작성예시`

```
SELECT /*+ 홍길동::com.team.project.member.model.dao.MemberDaoImpl.selectMember() */
       A.COL1
     , A.COL2
     , B.COL3
  FROM (SELECT C.COL1
             , C.COL2
          FROM TABLE2 C
         WHERE C.COL3 = '100'
       ) A
     , TABLE2 B
 WHERE A.COL1 = B.COL1
   AND A.COL3 = '111';
```

2.2.2. WHERE Clause

- 기술요령
  - `ALIAS` 명을 기준으로 한 라인에 하나씩 조건절을 기술하고 다음 라인에는 `AND`부터 기술한다.

작성예시

```
WHERE A.COL2 = B.COL2
  AND A.COL3 = B.COL3
  AND A.COL1 = '111';
```

  - 만약 서브쿼리가 있는 경우에는 `(`를 맞춰서 기술한다.

작성예시

```
WHERE A.COL1 = (SELECT B.COL1
                  FROM TAB2 B
                 WHERE B.COL2 = '111'
               );
```

  - `WHERE` 절 `DUMMY` 조건(`WHERE 1 = 1`)은 미사용을 원칙으로 하며 `Dynamic SQL` 사용이 승인된 프로그램에 한해서 예외적으로 허용한다.
- 기술 순서

1. `JOIN` 조건을 먼저 기술한다. (`JOIN` 되는 순서대로 기술한다.)
2. 상수 조건 및 범위비교 조건을 기술한다. (먼저 처리되어야 할 조건부터 순서대로 기술한다.)

3. 서브쿼리가 포함된 조건을 기술한다.
4. `ORDER BY` 나 `GROUP BY` 는 `WHERE` 조건이 아닌 별도 `LINE`에 기술한다.
    - `ORDER BY` 컬럼 나열 시 숫자가 아닌 컬럼명을 반드시 사용하도록 한다.
    - `ORDER BY`, `GROUP BY` 역시 컬럼 마다 줄 바꿈을 기본으로 한다.

```
WHERE A.COL1 = B.COL1
  AND A.COL2 = B.COL2
  AND A.COL3 = '123'
  AND B.COL3 = 'A12'
ORDER BY A.COL2
       , B.COL2;
```

2.2.3. SELECT Clause

- `SELECT LIST`에 `*`을 사용하지 않는다. 조회하고자 하는 컬럼만 나열한다.
- 컬럼 `ALIAS`의 지정은 컬럼명과 상이한 경우에만 사용하며 `AS` 키워드는 기술하지 않는다. (단, 함수, 서브쿼리, 컬럼 조합 연산 시에는 필수로 작성)
- 함수 사용으로 인해 기술 내용이 길어질 경우에는 이해가 용이하도록 적절한 줄바꿈과 괄호를 사용하도록 한다. (한 라인에 콤마가 나열될 시 콤마 앞에는 공백이 없고 콤마 다음에는 공백을 준다.)

`DECODE 사용 예시`

```
SELECT /*+ 홍길동::호출함수정보 */
       A.COL1
     , DECODE(SUBSTR(A.COL2, 1, 3), '111', B.COL2
                                  , '112', C.COL2, D.COL2) COL_ALIAS
```

`CASE문 사용 예시`

```
SELECT /*+ 홍길동::호출함수정보 */
       A.COL1
     , A.COL2
     , A.COL3
     , CASE
         WHEN A.COL3 LIKE 'A%' THEN '10'         -- 들여쓰기 2칸 임
         WHEN A.COL3 LIKE 'B%' THEN '20'
         WHEN A.COL3 LIKE 'C%' OR A.COL3 LIKE 'D%' THEN '30'
         ELSE '40'
       END VALUE_CODE
  FROM MYBAIS.TABLE1 A;
```

- 컬럼별 코멘트 작성 시 `/* */`의 형식으로 작성하고, 가장 길이가 긴 라인을 기준으로 정렬한다.

`컬럼별 코멘트 작성 예시`

```
SELECT /*+ 홍길동::호출함수정보 */
       A.COL1 || A.COL2    /* 컬럼1과 컬럼2 문자열합치기 */
     , A.COL3              /* COL3에 대한 정보 */
     , A.COL4              /* COL4에 대한 정보 */
  FROM MYBATIS.TABLE1 A;
```

2.2.4. UPDATE Clause

- 변경 컬럼이 두 개 이상 일 경우는 `,`로 나열하며 한 라인에 하나씩 기술한다.

```
UPDATE /*+ 홍길동::호출함수정보 */
       MYBATIS.TABLE1 A
   SET A.COL1 = '111'
     , A.COL2 = '222'
     , A.COL3 = '333'
 WHERE A.COL4 = '000';
```

- 서브쿼리를 이용하여 `UPDATE`를 하는 경우 `TABLE` 명에 `ALIAS` 명을 기술하여 컬럼의 모호성(`ambiguous`)이 발생하지 않도록 한다.
- `SET`절 서브쿼리의 결과값이 없는 레코드가 있는 경우 (`NULL` 반환) 의도치않게 `NULL`로 `UPDATE`될 수 있기 때문에 `WHERE`절에 동일한 `EXIST` 구문을 추가하거나, `MERGE` 구문(`UPDATE ONLY`)으로 변경되도록 한다. (`JOIN` 성능 상 `MERGE` 구문을 권고)

```
UPDATE /*+ 홍길동::호출함수정보 */
       MYBATIS.TABLE1 A
   SET A.COL1 = (SELECT SUM(B.COL2)
                   FROM MYBATIS.TABLE2 B
                  WHERE A.COL1 = B.COL1
                 )
 WHERE A.COL3 = '111'
   AND EXIST (SELECT 1
                FROM MYBATIS.TABLE2 B
               WHERE A.COL1 = B.COL1
             );
```

- 시스템 필수 컬럼 갱신을 누락하지 않도록 한다.

- 업무적인 요건에 따라 전체 컬럼을 `UPDATE` 해야 하는 경우를 제외하고 전체컬럼 `UPDATE`는 사용을 금지한다. `UPDATE` 되어야 할 컬럼을 식별하여 개별 `UPDATE`문으로 분리시켜야 한다.

`** 전체 컬럼 UPDATE 금지 사유`

- `CPU` 사용율 증가에 따른 성능 저하
  - 동일한 수의 블록이 `UPDATE`되지만, 블록 내에서 변경되어야 하는 컬럼 수가 많아짐에 따라 `CPU` 처리 시간이 늘어나면서 성능 저하 발생
  - 테스트 결과 : 단일 `ROW`를 백만 건 `LOOPING` 처리 시 약 `3배` 이상 성능 저하 발생
- `인덱스 갱신`에 따른 추가적인 성능 저하
  - `UPDATE` 대상이 아닌 컬럼에 생성되어 있는 인덱스도 불필요하게 갱신되어야 함게 따라 추가적인 성능 저하 발생
  - 테이블에 생성되어 있는 인덱스 개수에 비례하여 성능 저하 발생
- `REDO LOG SIZE` 증가
  - `DB` 변경 내용이 `TEXT 기반`으로 관리되고 있어 변경 컬럼 개수가 많아지면 `REDO LOG` 발생량이 많아짐에 따라 `DBMS` 내부적인 관리 비용이 증가
- 데이터 흐름 관리의 부정확한 결과 도출
  - 데이터 흐름 관리는 프로그램 소스 상의 테이블, 컬럼명을 기준으로 도출됨에 따라 `UPDATE` 대상 컬럼이 항상 전체가 되므로 부정확한 결과가 도출됨
- 컬럼 변경 시 영향 범위 확대
  - 특정 컬럼 변경시에도 전체 컬럼 `UPDATE`문을 사용하는 모든 프로그램에 영향을 주게 되므로 관리의 비효율성 발생

2.2.5. DELETE Clause

- `DELETE` 구문은 다음과 같이 기술한다.

```
DELETE /*+ 홍길동::호출함수정보 */
  FROM MYBATIS.TABLE A
 WHERE A.COL1 = '111'
```

- 업무적인 요건에 의해 의도적으로 전체 데이터를 삭제해야 하는 경우가 아니라면 `WHERE` 절을 반드시 기술하여 전체 데이터가 삭제되지 않도록 해야 한다.

- 서브쿼리와 연결하여 처리할 경우에는 TABLE 명에 ALIAS명을 기술하여 컬럼의 모호성(`ambiguous`)이 발생하지 않도록 한다.

2.2.6. INSERT Clause

- 향후 컬럼 순서 변경이나 추가 시 유연성을 갖고자 `INSERT` 되는 컬럼명을 필히 기술한다.

```
INSERT /*+ 홍길동::호출함수정보 */
  INTO MYBATIS.TABLE1 A
(
  COL1
, COL2
, COL3
, COL4
)
VALUES
(
  '101'
, '102'
, '103'
, '104'
);
```

- 컬럼 수가 5개 이상 많을 경우 컬럼과 입력되는 값의 위치와 갯수를 맞추어가며 가지런히 정렬한다.

```
INSERT /*+ 홍길동::호출함수정보 */
  INTO MYBATIS.TABLE1 A
(
  COL1, COL2, CON3, COL4, ...
, COL5, COL6, COL7, COL8, ...
)
VALUES
(
  '101', '102', '103', '104', ...
, '105', '106', '107', '108', ...
);
```

2.2.7. MERGE Clause

- 조건을 만족하는 `row`가 존재할 경우 `update`를 하고, 아닌 경우에는 `insert`를 수행한다.

- 대량 건 업데이트 수행 시 `Cursor`를 통한 `looping update` 보다는 일괄처리가 가능한 `merge`구문을 권장한다. (/*+ BYPASS_UJVC */ 힌트가 더 이상 지원되지 않는 `11g` 버전부터는 `Updatable Join View` 대신 `merge` 구문을 사용해야 한다.)
- SYNTAX
  - INTO : target table
  - USING : table, view, subquery 가능
  - ON : insert를 수행할 지 update를 수행할 지 결정할 때 사용되어지는 조건
  - WHEN MATCHED THEN : update 구문 작성
  - WHEN NOT MATCHED THEN : insert 구문 작성

```
MERGE /*+ 홍길동::호출함수정보 */
 INTO MYBATIS.TABLE1 A
USING (SELECT B.COL1
            , SUM(B.COL2) SUM_COL2
         FROM MYBATIS TABLE2 B
        GROUP BY B.COL1
      ) C
   ON A.KEY= B.KEY
 WHEN MATCHED THEN
 UPDATE
    SET A.COL1 = B.COL1 + B.COL2
 WHEN NOT MATCHED THEN
 INSERT
 (  A.COL1
  , A.COL2
 )
 VALUES
 (  B.COL1
  , B.COL2
 );
```

- 제약사항
  - 동일 `target row`에 대해 `merge` 구문 내에서 여러 번 `update`를 수행할 수 없다.

    ORA-30926 : unable to get a stable set of rows in the source tables
  - `ON` 절에서 사용된 컬럼은 `UPDATE`가 불가능하다.

    ORA-38104 : Columns referenced in the ON Clause cannot be updated.

2.2.8. 식별자

- `SQL` 모니터링의 용이성을 위해 어플리케이션 구축 시에 각 단위 프로그램마다 `SQL` 식별자를 반드시 기입하도록 한다.
- DML 키워드 오른쪽 같은 라인에 오라클 힌트 주석(`/*+ */`)의 형태로 기재한다.
  - SQL 식별자는 개발자명과 호출프로그램의 정보로 조합된다.
  - SELECT `/*+ 개발자명::호출프로그램정보 */` 의 형태로 작성한다.
  - 오라클 힌트 추가시에는 SQL 식별자보다 앞에 위치시키며 콤마`,`로 구분하여 하단에 SQL 식별자를 기입하도록 하며, 다수의 힌트를 사용 시 가독성 향상을 위해 힌트마다 줄바꿈을 권고한다.

```
SELECT /*+ HINT 구문
         , 개발자명::호출프로그램정보 */
```

3. 성능을 고려한 SQL 작성 지침

3.1. 리터럴 SQL

- 리터럴, 변수, 상수의 의미
  - 리터럴 : 컴파일 시 프로그램 내에 정의되어 있는 그대로를 정확히 해석되어야 할 값
  - 변수 : 프로그램의 실행 중에 상황에 따라 다른 값들을 표현할 수 있는 것
  - 상수 : 프로그램 실행 중 늘 같은 값
- 리터럴 SQL의 정의
  - 동일 SQL에 리터럴만 다르게 작성된 SQL
  - 조회 조건과 같이 입력되는 변수에 대해 상수값으로 하드코딩 되도록 개발된 SQL

```
SELECT ENAME FROM EMP WHERE DEPTNO = 10;
SELECT ENAME FROM EMP WHERE DEPTNO = 20;
```

- 리터럴 SQL 변수 처리 (바인드 변수 사용)
  - 모든 `SQL`문은 처리를 위해 반드시 파싱(`parsing`)과정을 거쳐야 하는데 이는 `CPU`를 많이 사용하는 작업이므로 파싱 오버 헤드를 줄이기 위해 커서를 공유할 수 있도

록 해야 한다.

  - 리터럴 `SQL`은 동일 `SQL`에 대해 반복적인 `parsing`을 유발하므로 반드시 변수를 바인딩하도록 한다. (`JDBC` 프로그램에서는 `PreparedStatement`를 사용)
  - 성능 향상의 목적과 업무적 특이성의 경우, 제한적으로 리터럴 `SQL`을 사용하며, 이는 `DBA`와 반드시 협의를 거쳐야 한다.

3.2. 반복 호출 제한

- 반복적인 단일 `SQL` 구문, `PL/SQL` 구문의 호출은 비례적인 응답속도의 저하를 발생시킨다.
- AP 서버와 DB 서버 사이간 반복적인 호출의 발생은 `network` 지연 요소까지 포함하여 성능저하가 보다 심하게 발생 가능하다.
- 설계와 개발 단계에서 준수되지 못할 경우, 이후 어플리케이션 수정에 많은 어려움이 발생하게 된다.
  - 반복적인 호출이 발생하는 경우
    - `PL/SQL(Procedure, Function)` `LOOP` 내부의 `SQL`구문
    - 다건 처리 시 `SELECT` 절, `WHERE`절에 사용된 `Function`
    - `JAVA` 소스 내부에서 `SQL` 구문, `PL/SQL` 의 반복 호출(`LOOP` 사용 등)
  - 반복 호출의 감소
    - `LOOP` 내부의 `SQL`구문은 `LOOP` 외부의 커서 등과 결합하도록 한다.
    - `LOOP` 를 이용하여 `insert`, `update`, `delete` 등의 `DML`의 수행은 일괄 처리 방식으로 변경하도록 한다.
    - `SELECT` 절의 `Function` 사용보다 `JOIN` 방식으로 해결 가능한지 우선 검토한다.
    - `JAVA` 소스 내부에서 반복적인 `DB` 작업 호출이 필요할 경우, `DB` 작업 결과를 1회 일괄로 수행하여 `AP`단에 저장한 후, 해당 결과를 이용하여 반복처리를 수행하도록 한다.

3.3. 처리 업무 제한

- 일반적으로 업무 처리 시 과도하게 많은 건수의 데이터 조회가 필요한 경우는 많지 않으나, 이에 대한 제약이나 화면 사용 가이드가 없을 시 많은 사용자들이 다량의 데이터를

조회하고자 한다.

3.3.1. 처리 범위 제한 방안

- 검색 범위 조정
  - 검색 범위가 넓거나 전체 조회를 하는 경우, 엑세스량 과다로 성능 저하가 발생하며 `SQL` 튜닝만으로 응답속도 개선에 한계가 있다.
  - 전체 조회를 막거나 조회 기간 최대 허용치를 정의하여 화면 단에 제약을 걸 수 있도록 한다.
  - 날짜 조건만 입력 시, 알림을 통해 추가 조건 필드 입력을 유도한다.
  - 적정 성능의 `DEFAULT` 조회 기간을 현재일 기준으로 조건 입력 필드에 `setting` 해두도록 한다.
- 검색 필드 조정
  - 필수적으로 입력해야 하는 필드를 사전에 정의하여 미입력 시 실행을 막도록 한다.
  - 필수 입력 조건이 아닐지라도 특정 `DEFAULT` 값을(현재일, 지역코드 등) 미리 입력 필드에 `setting` 하여 사용자가 조건 입력을 누락시키지 않도록 유도한다.
- 페이징 처리
  - 다건 조회 업무 요구사항인 경우, 페이징 처리를 통하여 많은 양의 데이터를 조회하기 위한 부담을 줄이도록 한다.
  - 사용자의 화면 활용 패턴에 따라 적합한 페이징 기법을 선택하도록 한다.
    - `next page`로 이동이 적고 이전 `page`로 이동이 필요한 경우 `rownum` 방식
    - `next page`로 이동이 빈번한 경우 `next key` 방식
    - `rownum` 방식
      - `SQL`문에 별도의 로직이 추가되지 않음
      - `F/W`에서 구현 기능 제공(중단 거래 방식)
      - 부분 범위 처리를 위해 `INDEX`재설계 및 `SQL` 튜닝이 필요할 수 있음
      - 최적화 시 첫 페이지는 빠르게 보여지나 다음 페이지로 넘어갈 수록 내부적인 `logical I/O`가 누적되어 증가됨
    - `next key` 방식
      - `SQL` 문에 `next key` 방식을 위한 `logic`이 추가됨
      - `F/W` 에서 구현 기능 제공(`scroll`방식)

- 부분 범위 처리를 위해 `INDEX` 재설계 및 `SQL` 튜닝이 필요할 수 있음
- 최적화 시 `page` 이동에 상관 없이 항상 일정한 건수만 부분범위 처리가 가능함

`paging SQL 샘플(페이지당 10건 보여지는 첫 페이지 조회 예시)`

`rownum`

```
SELECT *
FROM (SELECT INNER_TEMP.*, ROWNUM AS ALIAS
        FROM ([쿼리문 작성]
             ) INNER_TEMP
       WHERE ROWNUM <= 10 + 1
     )
WHERE ALIAS BETWEEN 1 AND 10 + 1;
```

`next key`

```
SELECT *
  FROM (
    SELECT ORG.*, ROWNUM
      FROM (
             [쿼리문 작성]
             ORDER BY N1, N2, N3 DESC
           ) ORG
     WHERE N1 >= :N1
       AND (N1 - :N1
        OR (N1 - :N1 AND N2 > :N2)
        OR (N1 - :N1 AND N2 - :N2 AND N3 <= :N3)
           )
        )
 WHERE ROWNUM <= 10 + 1;
```

3.3.2. ROW LIMIT Clause (12c new function)

- 오라클 12c 부터 제공되는 ROW LIMIT Clause는 성능 저하 및 결과값 오류 발생 등 아직 안정화 단계에 접어들지 않았다고 판단되어 사용을 금지하는 것을 원칙으로 한다.
- FETCH FIRST 1 ROW ONLY 구문을 오라클 환경으로 전환 시 INLINE VIEW & ROWNUM 방식으로 변경해서 사용해야 함

3.4 채번 설계

3.4.1 채번 유형

- ORACLE SEQUENCE
  - 장점
    - 가장 빠른 성능
    - insert 구문 내 직접 채번
    - lock wait 발생 없음
  - 단점
    - 결번이 발생하거나, 채번 순서가 역행될 수 있음
    - 타 컬럼에 종속적인 순번 생성 어려움(날짜 종속, 코드 종속)
- 최대값 + 1
  - 장점
    - 번호 연속성 유지됨
    - 업무에 맞춰 유연하게 활용 가능
  - 단점
    - 동시 트랜젝션 발생 시 데이터 중복 에러 유발 가능
    - 중복 에러 발생 예외 처리 필수
    - lock wait 발생 시 회피 방안에 한계점 존재
    - pk 컬럼과 맞지 않는 구조의 경우, 별도의 채번용 인덱스가 추가되어야 함
- 채번테이블
  - 장점
    - 번호 연속성 유지 가능
    - 복잡한 채번 패턴도
    - 프로그램의 변경 없이 유연하게 적용 가능
  - 단점
    - 채번 트랜젝션 올릴 시 lock wait가 빈번하고, 응답시간 저하가 예상됨
    - 결번이 허용될 경우 lock wait는 회피 가능

3.4.2. 채번 유형 선택 기준

- ORACLE SEQUENCE
  - 성능 최우선 업무에 사용
    - 동시 트랜젝션 집중 및 대량 배치 채번 유형의 업무에 필히 고려 함
  - NOORDER & CACHE 옵션 사용을 1순위로 함
    - 채번 유일성만 보장되는 요건으로 업무 협의 필요
    - ORDER나 NOCACHE옵션을 사용하는 경우 성능 개선의 이점이 없음
  - SEQUENCE 시작값 초기화는 DROP SEQUENCE가 필요한 작업으로 오브젝트 변경 영향도가 크다. 따라서 CYCLE 적용은 가능하지만 특정 시점을 정의하여 주기적인 재생성은 불가함을 원칙으로 한다.
  - SEQUENCE는 하나의 테이블에 종속되도록 설계하는 것을 원칙으로 함
    - 여러 테이블에서 공유해서 사용하는 것을 금지함
    - 오브젝트 명명 규칙 (SEQ_TABLENAME_01)
- 최대값 + 1
  - 복합 PK인 경우에 사용 권고
    - 타 PK 컬럼 그룹에 종속적인 순번 채번
  - 중복 에러 회피 가능 여부가 관건이다.
    - 채번 트랜젝션의 독립성이 완전히 보장되는 업무에만 사용
    - 업무적으로 동시 트랜젝션 발생 가능성이 있다면 반드시 오류에 대한 예외처리 필요함
  - 채번 방법
    - 단일 세션에 종속적인 채번의 경우 중복 가능성이 없으므로 "MAX(채번컬럼) + 1"방식을 사용해도 무방함. (SELECT FOR UPDATE 미사용)
    - 여러 세션이 동시에 채번할 가능성이 있는 경우 SELECT FOR UPDATE 구문을 NOWAIT모드로 수행하여 중복 에러 대신 ORA-00054(리소스 사용중)을 발생시켜 예외처리를 적용하도록 함. (WAIT 도중 LOCK을 획득한다 하더라도 중복 에러가 발생함)

    역스캔 힌트 방식

    ```
    SELECT /*+ INDEX_DEXC(A XBSQLA001P) */
           A.문서번호 + 1
    ```

```
  FROM MYBATIS.TABLE1 A
 WHERE ROWNUM = 1
   FOR UPDATE NOWAIT;
```

`MAX 함수 활용방식(표준안)`

```
SELECT A.문서번호 + 1
  FROM MYBATIS.TABLE1 A
 WHERE A.문서번호 = (SELECT MAX(B.문서번호)
                       FROM MYBATIS.TABLE2 B
                     )
   FOR UPDATE NOWAIT;
```

        - SELECT FOR UPDATE 구문은 ROW에 대한 LOCK이므로 집계 함수인 MAX 함수와 함께 사용 시 에러(ORA-01786)가 발생함
        - INDEX_DESC & ROWNUM = 1 방식으로 최대값을 찾는 경우 SELECT FOR UPDATE가 가능하다 HINT가 무용화되거나 INDEX에 문제 발생 시 프로그램 장애와 연결되므로 MAX 함수 서브쿼리를 사용한 WHERE절 PK JOIN 방식을 사용하도록 함.
- 채번테이블
    - 복합 PK & 코드 패턴 조합에 PK에 사용
        - 확장성 및 유연성이 감점임
        - 의미있는 코드 패턴의 한 구성으로 순번 채번이 필요할 때 사용 가능
    - LOCK 경합 회피가능 여부가 관건임
        - 결번이 허용 가능한 요건으로 업무 협의 필요
        - 결번 허용 불가시 경합 발생 가능성이 최소화되도록 트랜젝션 설계 필요
    - 채번테이블 트랜젝션 분리 방안
        - 결번 허용 가능 시 원 트랜젝션과 채번테이블에 대한 트랜젝션을 반드시 분리 구성한다.
        - 트랜젝션 PROPAGATION - REQUIERS_NEW

3.5 조건 분기

- Dynamic SQL은 개발은 수월할 수 있으나 성능 및 유지보수 문제점이 존재하기 때문에 기본적으로 쿼리 분리를 원칙으로 한다.

- 본 문서는 Dynamic SQL을 사용하지 않고 쿼리를 최소한으로 분리할 수 있는 기준을 제시한다.

3.5.1. 입력 조건 유형 별 분기 기준

- Dynamic SQL 사용 금지 사유
  - 데이터 흐름 관리 및 영향도 분석 솔루션에서 소스 기반 분석이 불가능하게 됨
    - 입력 조건에 따른 WHERE절의 동적 생성
    - FROM절 TABLE, SELECT절 컬럼명의 변수 처리
- 입력 조건에 따라 SQL을 분리해야 하는 이유
  - 입력 유형별로 쿼리를 명확히 분리함으로써 SQL 가독성, 유지보수성, 성능(SQL 튜닝 및 인덱스 설계)관리에 용이하도록 함
- 입력 유형 구분
  - 입력 유형에 맞추어 크게 3~4가지 정도로 쿼리 분리
  - 각각의 유형에 맞는 인덱스를 생성해야 성능 튜닝이 가능함
  - 선택적인 입력 조건 들 중 상태, 구분코드 등 콤보박스 형태의 선택 입력 값은 각각의 유형으로 분리할 필요 없음. (10개 내외의 공통 코드 유효값 수준)
    - 불균등한 데이터 분포에 따라 단독 혹은 다른 조건과의 조합으로 성능 개선이 가능한 경우도 존재함. 이 경우 DBA나 튜너의 검토 하에 분리 여부를 결정하도록 함

Aa 분리유형	Filter Factor Cardinality	인덱스효과성	입력컬럼(예)	거래빈도(예)
key 조회형(search)	높음	탁월	계좌번호 주민번호 고객번호 전문번호	일 1000만건
범위 조회형(scan)	중간	1일 : 우수 1개월 : 보통	지점코드 + 처리일자 부점코드 + 거래일자	일 10만건
리포트 조회형(집계)	낮음	낮음	넓은 일자 기간 조건의 전체 범위 처리 유형	일 1000건
제목 없음				

** 뒷부분은 내가 이해가 안갑니다

3.6 인덱스의 활용

- 다음과 같은 경우에는 인덱스가 존재해도 사용되지 못하므로 SQL 작성 시 주의하도록 한다.
  - 인덱스 컬럼에 외부적 변형이 가해질 때
  - 인덱스 컬럼에 내부적 변형이 가해질 때
  - 부정형으로 비교된 조건을 사용할 때 (≠, NOT IN)
  - IS (NOT) NULL로 비교 된 경우
    - 인덱스에는 NULL에 대한 정보가 없으므로 FULL TABLE SCAN이 발생한다.
  - LIKE가 DATE 또는 NUMBER TYPE의 컬럼에 사용된 경우
    - LIKE 연산자는 연산의 대상이 되는 컬럼이 CHARACTER TYPE이 아닌 경우 해당 컬럼을 CONVERSION 하므로 인덱스를 사용할 수 없다.
  - LIKE '%TEXT%' 또는 LIKE '%TEXT' 절
    - LIKE 조건에 '%'로 시작하는 비교값이 들어오면 인덱스를 사용할 수 없다.
  - 레코드 필터링을 위해서는 HAVING 보다는 WHERE를 사용한다. 인덱스가 걸려있는 컬럼에 GROUP BY와 함께 HAVING절을 사용하게 될 경우 HAVING절의 컬럼에 대한 인덱스는 사용되지 않는다. 즉 HAVING절은 GROUP BY절에 의해 이미 GROUPING된 데이터를 줄여주는 역할만 수행하므로 인덱스를 사용하지 않기 때문에 WHERE 조건에 의해 인덱스를 사용하여 데이터를 가져온 후 GROUPING을 하도록 한다.
  - 주요 조회 조건이 OR 로 분기되는 경우 양쪽 컬럼에 모두 인덱스가 생성되어 있지 않으면 FULL SCAN을 하게 된다.