# <데이터베이스 실습 복습 정리>

## 161010 to 161017(NOVALIDATE to JOIN)

NOVALIDATE는 제약 조건에 대해서 연관이 되니 제약 조건과 함께 참조하면 좋고, 이번 중간고사 범위인 Join 문까지 정리를 해두었다. Join문은 의외로 어려우니 한 번 정리를 해두고 공부하면 좋겠다.

---

## 9. Novalidate at 161010

우리가 제약 조건을 작성하게 되면서 언젠가는 이것이 필요로 없을 경우가 존재할 것이다. 그래서 Novalidate를 통해서 이 제약 조건을 제어할 필요가 있다.

Before) 연관된 조건에 관련해서 다루기

### 1) Disable Constraint

제약조건에 대해서 Disable(즉 제약조건이 원래는 있지만 적용이 안 되게끔 하는 개념으로 생각하자.)를 하는 경우가 더러 있다. 가령 예를 들어 값을 입력하는데 있어서 제약조건을 지키지 않고 추가하거나 값에 대한 삭제/갱신을 할 때 쓰는 경우로 생각하면 된다.

1-1) Disable Novalidate 제약_조건_이름

Novalidate 문장은 쉽게 이야기해서 제약조건이 본래 있는데 제약조건을 적용하지 않게끔 하는 문장이다. Constraint 실습문제 6번을 참조하자.

6. EMP3 테이블에 설정되어 있는 기본키 제한조건(EMP3_ID_PK)을 검사하지 않도록 설정하라.(1)
이 제한조건에 의존적인 다른 제한조건들도 모두 검사하지 않도록 하라.(2)
EMP3 테이블의 제한조건 정보를 확인하여 DISABLE되었음을 확인하라. DEPT3테이블의 DEPT_MANAGER_ID_FK 제한 조건도 DISABLE 되었음을 확인하라.

6-(1)

```
Alter Table Emp3
Disable Novalidate Constraint EMP3_ID_PK;
```

이를 작성하게 된다면 Emp3의 기본키 제약조건의 이름인 EMP3_ID_PK가 적용하지 않게끔 한다. 이처럼만 작성하게 된다면 EMP3의 기본키 컬럼인 ID가 null이 됐든 중복된 값이 됐든가 추가가 된다. 그렇지만 의존적인 다른 제한 조건에 대해서 검사하지 않도록 하려면 어떻게 해야 할까?

6-(2)

```
Alter Table EMP3
Disable Novalidate Constraint EMP3_ID_PK Cascade;
```

EMP3_ID_PK 뒤에 그냥 Cascade를 작성하면 된다. 의외로 간단하다. 이를 작성하게 된다면 현재 EMP3의 ID가 DEPT3 테이블의 Manager_ID라는 컬럼이 이를 참조하는 외래키로 설정이 되어있기 때문에 6-(1)처럼 작성하게 되면 EMP3의 테이블에 있는 제약조건인 EMP3_ID_PK만 삭제된다. 허나 이를 붙이면 문제 마지막 부분의 DEPT_MANAGER_ID_FK 이란 제약조건이 없어진다.

1-2) Disable Validate

Novalidate는 제약조건이 안 보이게끔 하는 것이지만, Validate는 제약조건을 방금 배웠던 Read Only와 같은 역할을 한다. 이를 설정하게 된다면 제약조건이 적용되지 않는 건 사실인데 데이터 갱신/삭제/삽입하는 것에 대해서는 제약조건을 본래대로 돌려놓지 않는 이상 실행되지 않게끔 한다. 만일 이를 설정하고 Insert Into 뭐시기를 써보면 이처럼 오류 문장이 나오게 된다.

```
Alter table EMP3
disable Validate Emp3_ID_PK;

Insert Into Table
Values(0, 'Boy', 'Girl', 10, 0.3);
```

-> 사용 안함으로 설정되고 검증된 제약 조건(HR.EMP3_ID_PK)을 사용하여 테이블에서 삽입/갱신/삭제 작업이 수행되지 않았습니다.

이는 쉽게 이야기해서 데이터에 대해서 변경하지 않게끔 하기 위해 작성된 문장으로 보면 된다.

### 2) Enable Constraint

Enable는 제약조건이 본래 없어 보이게끔 한 Disable과는 달리 원래대로 보이게끔 하는 개념으로 생각하면 된다. 역시 Novalidate, Validate 2가지로 나뉘게 되는데, 이 둘의 차이점에 대해서 알아둘 필요가 있다.

2-1) Enable Novalidate

이는 방금 Novalidate한 제약조건에 대해서 다시 검사를 하기 위해서 원상복귀 시키는 역할을 하지만 Disable Novalidate로 인해서 제약조건이 적용되지 않는 값에 대해서 무작위로 추가한 값에 대해서 차후에 Enable Novalidate를 하면 여태껏 저장된 값이 무엇이든 간에 신경을 안 쓰고 제약조건만 원상복귀 시키는 개념으로 생각하면 된다. 아래 문장을 참조하면 좋겠다.

```
Insert Into EMP3(ID, First_name, Last_name, Department_ID, Commission)
Values(null, 'ok', 'on', 10, 0.1); // 1
Alter table EMP3
enable Novalidate Constraint EMP3_ID_PK; // 2
select *
from emp3; // 3
```

본래 EMP3의 ID를 null로 저장할 수 없다는 점은 다 알고 있을 것이다. 왜냐면 본래 ID는 기본키라서 Primary Key는 Not Null, Unique 둘 다 융합한 개념으로 생각하는 것으로 봐서, 중복된 값과 null이 원래 들어갈 수 없기 때문이다. 허나 재미있는 사실은 2번 문장처럼 제약조건을 다시 살리고 난 뒤에 3번 문장을 실행해보면 값에 대해서 뭐가 됐든 간 검사하지 않고 그대로 출력하게 해준다. 허나 이처럼 작성하게 된다면 나중에 저장된 값들에 대해서 관리하는데 불편이 따를 것이니 아래와 같은 개념을 이용해보도록 하자.

2-2) Enable Validate

위 문장에서 1번과 같은 문장까지 작성하고 난 뒤 2번 문장을 다음과 같이 바꿔보도록 하겠다.

```
Alter table EMP3
enable validate Constraint EMP3_ID_PK;
```

enable Validate를 하게 된다면 제약조건을 다시 살리되 여태껏 저장된 값들에 대해서 제약조건에 위배된 값들이 있는지에 대해서 검사를 하고 난 뒤에 제약조건을 살릴 수 있다. 허나 중요한 것은 제약조건에 위배되는 값들에 대해서는 어떻게 처리를 해야 될까? 이는 바로 Exception 테이블이다.

1) 예외 테이블을 생성한다.
@?/rdbms/admin/utlexcpt.sql
이처럼 작성하게 된다면 예외 테이블 하나가 만들어지게 되는데 애석하게도 필자의 컴퓨터에는 실습이 안 되어서 책에 있는 내용만 쉽게 설명하고 넘기겠다. 여기서 ?는 $Oracle_Home 디렉토리다. 또한 utlexcep는 Utility Exceptions(예외 유틸리티)의 줄임말로 생각하면 된다.

2) 테스트용 테이블을 통해서 예외로 추가하기
Alter Table EMP3
Enable Validate Constraint Emp3_ID_PK
Exceptions Into exceptions;
이처럼 작성하면 에러는 걸리게 된다. 그렇지만 값은 그대로 지정이 되어 있는 것을 참조할 수 있지만 무엇보다 중요한 점은 예외 상황을 통해서 제약 조건에 어긋난 값들에 대해서 exceptions라는 테이블에 삽입을 하면서 결과는 오류가 뜨는 것이 정상이다. 그리고 마지막으로 예외 테이블을 조회하여 에러 내역을 확인하면 된다.

3) 예외 테이블을 조회하여 에러 내역 확인
Select RowID, ID, last_name, First_name,
department_ID, Commission
from EMP3
where rowid IN (Select row_ID from
Exceptions);
이처럼 작성하게 된다면 2번에서 실행된 조건에 대해 값들 중 올바르지 않는 값들을 RowID를 통해서 값들을 추가해 나가는 방식이다. 그래서 제약조건에 어긋난 데이터들에 대해서는 여기서 다루고 종결하게 된다.

### 3) Constraint 조회
제약조건은 다음과 같은 문장을 통해서 쉽게 조회가 가능하다.
Select owner, constraint_name,
constraint_type, status
from user_constraints
where table_name='EMP3';

여기서 owner는 사용자, constraint_name은 제약조건 이름, constraint_type는 제약조건 종류, status는 제약조건의 상태를 뜻한다.
- constraint_name은 처음에 제약조건을 추가하지 않은 경우 임의의 값이 나오게 되어서 숙지하기 힘들게 되니 나중에 제약 조건을 추가하는 경우에는 이름을 꼭 추가하길 바란다.
- Constraint_Type는 쉽게 말해 제약조건의 종류이다. P는 아시다시피 기본키, R(eferences)는 외래키, U는 Unique, C는 Not Null를 뜻한다.
- status는 방금 Disable, Enable에 대해서 출력을 하는 것이다.

### 4) 테이블의 정보 출력하기
select table_name, column_name,
constraint_name
from user_cons_columns;
이처럼 작성하게 되면 지금 테이블에 저장된 값들과 컬럼들, 제약조건들에 관련해서 모두 출력하게 해준다. 그래서 나중에 여러분들이 테이블에 어느 테이블과 컬럼, 제약조건들에 대해서 출력하기 위해서는 user_cons_columns라는 테이블을 알아두길 바란다.

---

## 10. Join at 161012
Join 문장은 각 테이블에 값들이 따로 띄어져 있을 때 쓰면 제일 유일한 문장이다. 오라클은 RDBMS(관계형 데이터베이스)의 개념이기 때문에 테이블이 몰려 있는 것이 아닌 테이블 별로 정규화 되어 있다. 그래서 Join 문을 쓸 때에는 컬럼들이 각각 다른 테이블에 흩어져 있을 때 Foreign Key를 이용해서 이용하면 좋다. 물론 질의 최적화기(query optimizer)를 통해서 이 문장에 대한 성능을 보장해주는 역할을 해준다. 여기서 알아둬야 할 점은 Join도 또한 쿼리(질의)라는 점을 알아둬야 한다. 물론 지금 이용하는 Oracle에도 적용되거나 다른 SQL에서도 적용이 가능한 표준 Join 방법이 있다.

1) Cartesian Product(Cross Join)
10곱하기 10은 100이다. 튜플 10행과 튜플 10행은 100행이 출력된다. 여기서 카티션 곱은 Join 쿼리 중에서 Where 절에 기술하는 Join 조건이 잘못 기술되었거나 아예 없을 경우 발생하는 경우이다. 우선 Cross Join을 이용해서 아래와 같은 문장을 작성해 보겠다.
select e.last_name, d.department_name
from employees e cross join departments d
where e.department_id=30 and d.department_id
between 30 and 50;
우선 여기서 On 문장을 생략하였지만, 2번부터 On을 까먹으면 답 없으니 생략하지 말도록 하자. 우선 e(사원 테이블)의 department_id가 30인 직원은 총 6명이다. 그리고 d(부서 테이블)의 id가 30, 40, 50이라서 부서는 총 3개다. 이를 곱하면 결과는 총 18개의 튜플이 나오게 된다.

2) Equi Join (on 문을 활용하기)
가령 예를 들어 사원의 직급(Job_ID)에 대해서는 다 알고 있지만, 직급명 해설(Job_Title)에 대해서 알고 싶을 경우가 있다. 그래서 ON 문장을 이용해서 다음과 같이 작성해 보겠다.
select e.last_name, j.job_title
from employees e join jobs j // 1
ON e.job_id=j.job_id; // 2
1 : 여기서 employees와 jobs에 무언가가 보일 것이다. 바로 소문자로 적혀 있지만, 이는 테이블 이름을 대신하는 Alias 개념으로 생각하면 된다. 물론 안 써도 되지만, 컬럼에 테이블 이름써야 하는 불상사가 일어날 수 있으니 이건 여러분들의 선택에 맡기겠다.
2 : ON 문장은 대개 각 다른 테이블에 같은 데이터가 존재하는 경우(Job_id, Department_id 같은 거, 즉 외래키 같은 경우)에 쓰이지만, 물론 외래키가 아니어도 On 문장을 쓰는데는 문제가 없다. 왜냐면 다음에 배울 Non-Equi Join을

참조해보자. 이처럼 외래키끼리 묶어주게 된다면 Jobs 테이블과 Employees 테이블이 서로 연결 고리를 형성해서 외래키를 통해서 참조되는 부모 테이블 외래키로 참조하는 새끼 테이블의 컬럼을 이용해서 부모 테이블의 컬럼을 불러올 수 있다.

3) On 문장에 조건을 이용하기
다.음.쪽.참.조.
1번 문장과 2번 문장의 차이를 알아보자.
(1)
```
select e.last_name, j.job_title
from employees e join jobs j
ON (e.job_id=j.job_id and e.job_id like 'SA\_%'
escape '\');
```

(2)
```
select e.last_name, j.job_title
from employees e join jobs j
ON (e.job_id=j.job_id)
where e.job_id like 'SA\_%' escape '\';
```

1번과 2번의 문장에 대해서 설명하면 직원의 job_id가 SA로만 시작하는 경우(_제외하고)에 대해서 출력을 하는 것이 문제이다. 허나 1번과 2번의 문장 결과는 모두 35행으로 결과는 똑같다. 차이점이 무엇일까? 바로 속도이다. 속도.
(1)번과 같이 작성하게 되면 ON 문장을 통해서 검색 조건부터 먼저 검사하여 데이터를 골라 낸 뒤에 join을 수행하게 된다. 허나 (2)번처럼 작성하게 된다면 ON 문장이 끝나고 난 뒤에 또 Where 문장을 작성하게 되면 (1)번에는 이미 ON 문장을 통해서 정리가 완료되어 금방 끝나지만, Where에서 107행(이려나...)에 대해 재확인하기에 오래 걸리게 된다. 그래서 (1)번과 같이 작성하면 속도 하나는 장담할 수 있으니 참조하자.

4) Non-Equi Join(Between 문장 활용)
```
20001   30000   날라리사원
10001   20000   싸가지사원
5001    10000   개간지사원
0       5000    신입사원
Max     Min     Title
```
다음과 같이 최대값과 최솟값을 범위로 둬서 직원들이 봉급에 따라서 어느 타이틀에 속하는지에 대해 작성하는 문장을 작성해 보겠다. 테이블 이름은 Salary_Result이다.

```
select e.last_name, e.salary, sr.title
from employees e join salary_result sr
ON e.salary Between sr.min_salary and
sr.max_salary;
```
우리가 방금 ON 문장에서는 대입 연산자만 써왔지만, Between 문을 통해서 같은 값이 아니더라도 그 값에 대해서 크고 작음을 따져 나가면서 그 사원이 어느 타이틀에 속하는지에 대한 출력문장을 작성해 봤다. 이처럼 작성하면 봉급(salary)이 20000달러 초과는 날라리사원, 10000달러 초과는 싸가지사원... 이런 개념으로 생각하면 된다.

5) Outer Join
가령 예를 들어 사원 중에 부서에 속하지 않는 경우가 있을 것이다. 이를 구분하기 위해서 Outer Join을 이용할 필요가 있다. 이는 왼쪽 테이블에 값이 없는 경우, 오른쪽 테이블에 값이 없는 경우, 양쪽 둘 다 없는 경우 3가지로 나누어서 작성할 수 있다.

5-1) Left Outer Join
```
select e.last_name, d.department_name
from employees e left outer join departments d
on e.department_id=d.department_id;
```

여기서 last_name이 Grant라는 녀석이 부서에 속하지 않은 경우가 있을 것이다. 여기서 left outer는 좌측에 있는 값들 중에서 우측에 없는 값(즉 사원 중에 부서가 없는 사람에 대한 정보를 출력)도 또한 출력하게 된다. 그래서 결과는 107행인가 모두 나오게 된다.

5-2) Right Outer Join
```
select e.last_name, d.department_name
from employees e right outer join departments d
on e.department_id=d.department_id;
```

밑줄 친 부서 테이블에 속하지 않은 값에 대해서는 출력을 하지 않게 하는 문장을 통해서 우측에 속하는 테이블에 대해서는 출력하되 좌측에는 없는 값도 출력하게끔 하는 문장이 바로 right outer join이다. right Outer를 이용하게 되면 departments 중에서 사원들이 아예 속하지 않은 부서가 존재하는 경우를 통해서 모든 컬럼을 출력하되 사원들이 속하지 않은 부서이름도 출력하게끔 해주는 문장을 살펴봄으로서 우측 테이블에 값이 존재하나 좌측에는 없는 것을 이용하는 것이 바로 right Outer 문장이다.

5-3) Full Outer Join
```
select e.last_name, d.department_name
from employees e Full outer join departments d
on e.department_id=d.department_id;
```

이 문장은 쉽게 말해서 5-1과 5-2를 Union 한 문장에 이름도 없고 부서도 없는 값들까지 포함한 결과로 생각하면 된다. 그래서 결과는 이것이 더 많이 나오게 된다.

(참조1) left Outer와 Right Outer 둘 다 햇깔리는 경우가 있는데 join 문장을 통해서 보면 아래의 기준을 통해서 햇깔리는 일이 없도록 하자. 빈 칸은 각각 아래에 적혀둔 키워드를 넣으면 된다.
from 좌측_테이블 (  ) join 우측_테이블
- Left Outer : 좌측에는 값이 존재하나 우측에는 값이 없는 경우도 출력
- Right Outer : 우측에는 값이 존재하나 좌측에는 값이 없는 경우도 출력
- Full Outer : 값이 둘 다 없는 경우에도 출력
Left Outer/Right Outer에도 출력된 각각의 값들도 모두 출력하게 된다.

(참조2) Outer Join에 대해서는 모든 값에 대해서 출력하는 것이 좋아지지만... 사실 자칫 잘 못 하다가 SQL에 안 좋은 영향을 끼칠 수 있다. 왜냐면 인덱스가 있더라도 이를 쓰지 않고 Full Scan을 하기 때문에 Outer Join은 웬만하면 적당히 쓰는 것이 좋다.

6) Self Join
우리가 C언어에서 재귀 함수를 통해서 팩토리얼,

제곱 등등을 다뤄본 적이 있었을 것이다. 이처럼 테이블도 자신의 다른 테이블을 참조하는 경우가 있다고 생각하면 된다. 가령 예를 들어 흔히 실습 때 배우는 Manager_ID를 생각하면 된다. Manager_ID는 현재 Employee_ID의 선배 혹은 선임으로서 Employees 테이블에도 Manager(선배/선임)의 정보도 또한 포함이 되어 있다고 가정을 하면 부하의 선임 정보들에 대해 알고 싶다면 이 문장을 작성해 보는 편을 추천한다.

다음 문장은 각 사원별 선임과 선임 전화번호를 출력하는 문장이다.

```
select e.last_name, e.phone_number,
m.last_name, m.phone_number // 1
from employees e join employees m // 2
ON(e.manager_id=m.employee_id); // 3
```

2 : 이처럼 같은 테이블에 다른 Alias를 통해서 참조하는 방법이 바로 Self Join의 개념이라고 생각하면 된다. 허나 주의할 점은 같은 Alias로 하면... 차라리 employees만 쓰도록 하자.

3 : 사원의 manager_ID(즉 선임/선배 직원 번호)를 선임/선배 정보의 employee_id를 참조하게끔 해서 선임/선배의 각 정보에 대해 참조하게끔 해주는 문장이 바로 ON이다. 부하의 선임/선배 번호가 곧 재귀 테이블인 m을 통해서 선임/선배의 정보가 출력하게 되는 원리로 생각하면 된다.

이번에는 사원번호가 126번인 직원에 대해서 같은 부서에 근무하는 사원들에 대한 정보를 출력하는 문장을 작성하겠다. 여기서 126번 직원에 대한 정보는 제외하고 오로지 같은 부서에서 일하는 직원들만 출력한다.

```
select c.employee_id, c.last_name,
c.department_id
from employees e join employees c
on(e.employee_id<>c.employee_id)
where e.employee_id=126 and
e.department_id=c.department_id;
```

여기서 e.employee_id<>c.employee_id를 하게 되면 내 자신 이외에 직원들에 대해서 참조하겠다는 뜻이다. 그래서 이처럼 작성을 하게 된다면 직원 번호가 126번인 직원에 대해서 같은 부서에 일하는 동료들에 대해서 출력하는 것으로 종결이 된다.

---

## 11. Sub Query at 161017

서브 쿼리는 쉽게 이야기하면 where 문에 select~from~where가 들어갈 수 있고, select 문에도 select~from~where가 들어갈 수 있고 여러 경우가 있다.
가령 예를 들어 최모 씨랑 같은 동네 사는 사람이 누구인가, 아니면 이번 달 봉급이 제일 많은 사람이 누구인가에 대해서 출력하는 경우가 있을 것이다. 그리고 다중 행에 대해 다루는 join 문장에 대해서 짤막하게 줄어주는 큰 역할을 하는 문장이 바로 서브 쿼리이다. 우선 서브 쿼리의 구조는 어떻게 되어 있는지 알아보도록 하자.

1) Sub Query 구조
우측 상단부를 참조하도록 하자.

```
Select employee_id, last_name
from employees
where 조건연산자 (Select 컬럼들
                  From 테이블_이름
                  Where 조건);
```

우선 밑줄 친 부분이 바로 진짜 Sub Query라고 하는데, Inner Query라고도 한다.
그리고 외부에 있는 쿼리는 Main Query(혹은 Outer Query)라고도 한다. 서브 쿼리의 주의 사항에 대해서 여기에 필히 추가하였다.
<> 서브 쿼리는 Where 문 우측에 항상 위치해야 하는 점이고, 괄호로 꼭 묶어줘야 된다.
<> 특별한 경우를 예외하고 Sub Query 내부엔 Order By 절 자체를 쓸 수 없다.
<> 단일 행 Sub Query, 복수 행 Sub Query 행 2가지로 나뉘어서 여러분들이 잘 선택해서 쓰길 바란다.

1-2) 서브 쿼리 기본 활용법
서브 쿼리의 종류로서는 4가지로 분류를 할 수 있다. 이 4가지를 중점으로 해서 알아보도록 하자.
- 단일 행 서브쿼리
- 다중 행 서브쿼리
- 다중 컬럼 서브쿼리
- 스칼라 서브쿼리

A) 단일 행 서브쿼리
단일 행을 통해서는 우리가 유일한 값에 대해서 추출하는 것을 알고 있다. 그래서 Sub Query의 수행 결과를 살펴보면 1건만 나오고 이를 Main Query로 전달해서 수행하게끔 한다. 단일 행 서브쿼리에서는 우리가 평소에 쓰던 연산자(=, !=, <>, >, <, >=, <= 등등)를 쓸 수 있으니 참조하도록 하자.

가령 직원 이메일이 RMATOS(이메일로 한 이유는 이름이 동명인 경우에는 단일 행이 아닌 다중 행으로 나오기 때문에 다시 생각을 해봐야 한다.)인 직원이랑 부서가 같은 직원들의 직원번호, 성, 봉급에 대해 출력하는 문장을 작성해 보겠다.

(1) 일반 행에서 이용하기

```
Select Employee_ID, Last_Name, Salary
from employees
where Department_ID=(Select Department_ID
                     From employees
                     Where EMail='RMATOS');
```

이처럼 작성을 하게 된다면 우선 RMATOS가 이메일인 직원의 부서번호가 50이 나온다. 서브 쿼리의 내용을 해결하면 밑줄 친 부분이 Where Department_ID=50으로 바뀌는 것으로 살펴보면 되고 결국에는 사원 번호가 50번인 직원들이 나오게 된다.

(2) 복수 행 함수에서 이용하기
우리가 지난 번에 복수 행 함수에 대해서 배웠을 것이다. 전체의 값에 대해서 합계를 내거나 평균을 낸다면 결과 값은 하나만 나오는 것을 이미 알고 있을 것이다. 이를 이용해서 봉급이 가장 적은 직원이 누군지에 대해서 출력하는 문장에 대해 작성해 보겠다.(컬럼은 (1)번처럼 나왔다고 친다.)

```
Select Employee_ID, Last_Name, Salary
from employees
Where Salary=(Select Min(Salary) From
Employees);
```
이처럼 작성하게 되면 우선 괄호 내부에 봉급이 제일 적은 값에 대해서 출력이 되고, 해결된다면 전체 문장에 대해 해결을 하게 되므로 봉급이 가장 적은 직원의 정보가 나오게 된다.

B) 다중 행 서브 쿼리
서브 쿼리를 실행하게 되면 결과가 여러 개가 나오는 경우가 있다. 그래서 평소에 작성하는 조건 연산자에 대해서는 문장을 실행하는데 있어서 무리가 있다. 그래서 다른 연산자가 있는데 아래를 살펴보자.
Where 컬럼 IN 서브쿼리
IN() 함수에 대해서는 이미 알고 있을 것이다. 서브쿼리를 통해서 나온 값들을 IN으로 해서 그 값에 속하는 자료에 대해 출력하는 문장으로 많이 쓰이니 알아두도록 하자.

Where 컬럼>ANY(서브쿼리)
여기서 Any라는 뜻이 모든, 다소라는 뜻으로 판단이 되지만, 제일 작은 값을 이용해서 비교가 되어 최솟값을 반환하게 된다.

Where 컬럼<ANY(서브쿼리)
이는 반대로 최댓값을 반환하게 된다.

Where 컬럼>ALL(서브쿼리)
여기서 All은 모든이라는 뜻으로서 모든 값에 대해서 큰 값을 반환하니 최댓값이 반환하게 된다.

Where 컬럼<ALL(서브쿼리)
이는 반대로 최솟값을 반환하게 된다.

쉽게 외우기 위해서는 밑줄 친 부분을 기준으로 해서 부등호를 반대로 바꿔두면 개념도 반대로 바뀐다는 개념으로 생각해서 외워보면 편하다. 기준으로 밑줄 친 부분에 대해서 알아두면 편하게 공부할 수 있으니 참조하자.

그래서 필자가 햇깔리는 경우를 대비해서 사원번호가 110인 경우에는 봉급이 각각 8300, 12008원이 나올 것이다. 그래서 위의 문장을 이용해서 설명을 하겠다.

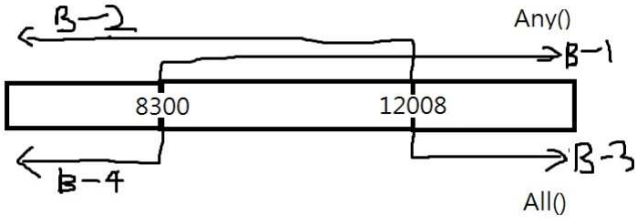
B-1) >ANY
```
Select employee_id, department_id, salary
from employees
where salary>ANY(select salary from
employees where department_id=110);
```
여기서 >Any() 부분을 하게 되면 최솟값을 반환하게 되어서 봉급이 8300보다 커지는 사원들에 대해서 출력을 하고 종결한다.

B-2) <ANY
```
Select employee_id, department_id, salary
from employees
where salary<ANY(select salary from
employees where department_id=110);
```
여기서 <Any() 부분을 하게 되면 최댓값을 반환하게 되어서 봉급이 12008보다 작은 사람들에 대해서 모두 출력하게끔 해주는 역할을 하고 종결 한다.

B-3) >ALL
```
Select employee_id, department_id, salary
from employees
where salary>ALL(select salary from employees
where department_id=110);
```
여기서 >ALL() 부분을 하게 되면 최댓값을 반환하게 되면서 봉급이 12008보다 큰 사원들에 대해서 출력하는 경우이다.

B-4) <ALL
```
Select employee_id, department_id, salary
from employees
where salary<ALL(select salary from employees
where department_id=110);
```
여기서 <ALL() 부분을 하게 되면 최솟값을 반환하게 되면서 봉급이 8300보다 아래인 사원들에 대해서 출력하는 경우를 보여준다.

위의 식에 대해서 햇깔리는 경우가 많이 있어서 필자가 그래프로 범위에 대해서 표현을 했으니 참조하길 바란다.




(필자가 대충 그렸지만... 이해해주길 바라면서...)

C) 다중 컬럼 서브 쿼리
서브 쿼리도 여러 개의 컬럼을 이용해서 비교를 할 수 있다. 허나 중요한 점은 컬럼의 종류도 종류별로 구분하는 것과 개수에 대해서도 정확히 적어둬야 적용이 된다. 아래 문장을 살펴보자.
```
Select employee_id, last_name
from employees
where (department_id, hire_date) IN
(Select department_id, MIN(hire_Date)
from employees group by department_id);
```
이 문장을 쉽게 설명하자면, 우선 직책 번호 별로 입사 날짜가 제일 오래된 직원들에 대해서 department_id와 Hire_Date에 대해 IN으로 각각 내용이 저장된다. (가령 20번 부서가 2015년 1월 3일, 30번 부서가 2016년 2월 29일...) 이를 만족하는 직원들에 대해서 조건을 따르게 되어서 직원 이름과 직원 번호를 출력하는 것으로 끝낸다.

D) 스칼라 서브 쿼리
스칼라 서브 쿼리는 Join 문장처럼 e.employee_id처럼 써먹을 수 있는 문장으로서 한 문장에 하나의 튜플이 나오게끔 하면 언제든 쓸 수 있는 개념이다. 아래의 문장에 대해서 참조 해보자.

```
Select e.employee_id, e.last_name, (Select
d.department_name from departments d where
d.department_id=e.department_id)
"Department_Name"
from employees e;
```

이를 살펴보면 우선 필자는 문장의 완전 보장성을 위해서 테이블 별로 키워드를 추가하는 방식으로 작성을 하였지만, 다음과 같이 써도 아무런 문제가 되지 않다는 점도 알아두면 좋겠다.

Select employee_id, last_name, (Select department_name from departments d where d.department_id=e.department_id) "Department_Name"
from employees e;
허나 주의해야 할 점이 있다. 스칼라 서브쿼리에서는 한 행만 출력해야 하기 때문에 on 문장을 쓰게 된다면 스칼라 서브쿼리 부분이 모든 행이 나오는 결과로 끝내기 때문에 ON을 쓰지 않는 편이 좋겠다.

그럼 여기서 스칼라 서브 쿼리는 어떻게 적용이 될까?
- 메인 쿼리(즉 우리가 흔히 쓰는 컬럼들부터 시작)부터 수행을 하고 스칼라 서브 퀄리에 대해 필요한 값을 수행한다.
- 스칼라 서브 쿼리를 수행하기 위해서는 필요한 데이터가 들어있는 블록을 메모리로 로딩을 하고,
- 메인 쿼리에서 주어진 조건(방금 d.department_id=e.department_id 이거 말하는 거다.)을 가지고 필요로 한 값을 찾아 쓰는 거로 한다. 다음에는 이 결과를 메모리 내의 Query Execution Cache(쿼리 실행 캐시)에 저장을 해둔다. 여기서 입력은 메인 쿼리 결과 값, 출력 값은 스칼라 서브 쿼리 결과 값이다.
- 메인 쿼리에서 스칼라 서브 쿼리로 들어오면 해쉬 함수를 이용해 해당 값이 캐쉬에 존재하면 즉시 결과 값을 출력하고 없으면 다시 블록해서 해당 값을 찾은 뒤 다시 메모리에 캐쉬를 하고, 이 작업을 메인 쿼리가 끝날 때까지 반복을 한다.

스칼라 서브 쿼리는 속도가 빠른 이유가 찾는 데이터가 메모리에 만들어져 있는 값을 찾아 오는 이유이기도 하지만, 모든 데이터가 없거나 데이터 양이 많은 경우에는 쿼리 실행 캐시에서 해당 데이터를 찾는 시간이 오래 걸려서 어쩌면 JOIN 문장보다는 느릴지도 모르겠다. 그래서 필자가 상황에 맞게 쓰는 점도 또한 추천하는데 데이터가 무지막지하면 Join 문장을, 참조하는 데이터가 그닥 많지 않으면 스칼라 서브 쿼리 문장을 추천하고 싶다.

---

## 12. INDEX at 161017

필자가 생각한 바로서는 인덱스, 서브 쿼리, JOIN 문장에 대해서는 본론을 꼭 읽고 시험에 임하길 바란다. 왜냐면 여기서는 문장 작성보다는 객관식으로 내거나 서술형으로 출제할 가능성도 살펴보고 있기 때문이다. 그래서 인덱스에 대해서도 알아둘 필요가 있으니 한 번 알아보자.

1) 인덱스의 정의와 원리
예를 들어 친구들이랑 약속을 잡을 경우에 '오후 3시까지 홍대입구역 3번 출구에서 만나자'(1) 라고 명시를 거의 하지만, 그렇다고 '오후 3시까지 서울시 마포구 서교동 무슨 대로 무슨 번지 무슨 건물에서 만나자'(2) 이런 식으로 약속을 명시하진 않는다. 이 문장에서 알 수 있듯이 우리가 쓰는 테이블 내에 튜플들에 대해서는 (2)번처럼 한글이 아닌 알파벳을 통해서 저장(ROWID라고 하는데 주소에 대해서 굳이 해석할 필요까지는 없다.)이 되어 있어서 순서는 임의대로 지정이 되어 있어서 데이터의 값들이 많아지는 경우에는 Table Full Scan을 통해서 계속 읽어야 하기 때문에 시간이 오래 걸리게 된다. 그러나 사용자가 데이터에 대한 접근성이 자주 있거나 join 문장 등을 통해서 많이 활용하는 경우에는 (1)번처럼 우리가 흔히 모이는 장소로 약속을 잡듯이 흔히 쓰는 ROWID를 흔히 찾는 인덱스의 이름으로 따로 설정하는 과정을 통해서 사용자가 원하는 컬럼이나 튜플들에 대해서 빠른 참조를 돕는다. 이것이 바로 인덱스의 원리이다.

2) 인덱스의 생성 원리
인덱스의 생성 원리는 B-Tree이든 Bitmap 이든 둘 다 비스무리하다. 그렇지만 우리가 인덱스를 만드면서 해당 데이터들이 변경하지 못 하도록 조치를 한 후 메모리(Sort Area에서)에서 정렬이 되는데 이 과정이 오래 걸린다. 그래서 쿼리(Join)의 속도를 향상하기 위해서는 정렬을 최대한 줄여야 하는 점을 목표로 두고 있다. 그리고 인덱스 생성 과정은 전체 테이블 스캔->정렬->블록 기억이라는 과정을 통해서 형성이 된다는 점을 알아두자.

3) 인덱스 구조와 작동 원리
인덱스는 컬럼이 인덱스의 영향을 받은 컬럼(Key)과 ROWID 2개 밖에 없다. 물론 영향을 받은 컬럼이 2개 이상일 땐 상황이 달라지니 참조하자. 이는 오름차순으로 생성이 되어서 만일 사용자가 이름이 '박효신'이라면 '박효신'이 저장된 ROWID를 통해서 해당되는 블록만 찾아서 복사를 해온다.

4) 인덱스의 종류
B-Tree, BITMAP 대개로 2가지가 존재한다.
OLTP에서는 대량의 데이터를 한꺼번에 추가하는 경우 사용하는 환경이라서 B-TREE 인덱스를 주로 이용한다. OLTP(Transaction)가 실시간 트렌잭션 처리용으로서 우리가 지난번에 트랜젝션에 대해서 공부한 적이 있을 것이다. 이를 이용한 원리로 데이터에 대해서 값을 추가/삭제/갱신하는 경우로 생각을 하면 된다. 반대로 OLAP(Analytical)는 온라인에서 실시간으로 입력되고 수정되는 경우에 쓰는 경우이다. 우리는 OLTP에서 주로 쓰는 인덱스를 참조해서 알아보도록 하자.

5) B-Tree Index
1. Unique Index
Key 값에 중복되는 데이터가 없다는 뜻으로, 이가 설정되어 있다면 해당 테이블의 컬럼에 중복된 값이 없다는 뜻으로 앞으로도 중복된 값은 들어올 수 없다는 점이라고 생각하면 된다. 가령 이름같이 중복된 값이 추가되는 경우에는 쓰지 말자.
Create Unique Index Idx_Depart_Name
ON Departments(Department_Name (DESC/ASC));
물론 DESC와 ASC를 통해서 내림차순/오름차순을 통해서 인덱스를 지정을 할 수 있으니 참조하도록 하자.

2. Non Unique Index
Create Index Idx_Depart_Name
ON Departments(Department_Name);
방금 중복된 값들이 들어오는 경우에 참조하는 인덱스는 바로 이거를 쓰길 추천한다.

3. 함수 기반 인덱스
Create Index Idx_Emp_Salary
On Employees(salary+10000);
가령 사원들이 추석이나 설날로 봉급날에 떡값을 받는 경우가 있을 것이다. 그래서 봉급에 대한 인덱스를 참조하도록 해야 되는데, 여기서 봉급에 함수에 대한 값을 저장해서 사용자가 떡값을 받은 봉급에 대해서도 빠른 탐색을 도와준다. 허나 이는 기존에 salary에 관련된 인덱스에 대해서는 참조하지 않는 개념이니 조건이 변경되면 또 다시 만들어야 하는 노가다가 따르기 마련이다.

4. 내림차순 인덱스
Create Unique Index Idx_Depart_Name
ON Departments(Department_Name DESC);
이는 큰 값을 주로 다루는 인덱스에서 쓰길 바란다. 물론 작은 값을 주로 다루는 인덱스에는 오름차순 인덱스(1번 참조)를 사용하길 바란다. 허나 오름차순 조건과 내림차순 조건에 대해서 융합을 해서 작성을 하게 된다면 DML 성능에 악영향을 미치게 되니 되도록 오름차순은 오름차순, 내림차순은 내림차순 이렇게 쓰길 바란다.

5. 결합 인덱스
Create Index IDX_Emp_DeptID_Salary
On Employees(department_id, salary);
2개 이상 컬럼을 작성한 인덱스를 결합 인덱스라고 한다. Where 절에서 And 연산자에서 많이 쓰이니 참조하도록 하고, 대개 많이 쓰는 인덱스의 종류이다. 가령 직무 번호가 총 10개 있고 salary의 값이 20개 존재하는 경우에는 먼저 직무 번호를 탐색하고 난 뒤 salary를 탐색하는 경우로 해서 출력을 하게 된다.

6. B-Tree Index/Bitmap Index
B-Tree Index는 주로 OLTP(지금 우리가 배우는 트랜젝션 오라클)에서 쓰고, Bitmap Index는 OLAP에서 많이 쓰이게 된다. Bitmap은 B-Tree Index와는 달리 중복된 값들과 수정이 종종되는 데이터에 대해서 참조하는데 있어서 많이 써먹는 인덱스로 살펴보면 된다.

7. 인덱스의 주의사항
1. DML에 취약하다.
- Insert 문에서는 Index Spilt 현상이 발생할 수 있어서 작업의부하가 심해지게 된다. Index Spilt는 인덱스의 블록들이 하나에서 2개로 나눠지는 현상으로서, 여유공간이 없는 경우에서 새로운 데이터가 입력이 되는 경우에는 성능 저하가 될 수 있다는 점을 알아두길 바란다.
- Delete 문에서는 테이블의 내용은 지워지지만 인덱스는 그대로 남아있는 대신 사용할 수 없게 막아둔다. 이를 사용하게 된다면 인덱스를 사용함에도 불구하고 오히려 느려진다.
- Update 문에서도 위험하다. 물론 Update 문에서도 인덱스가 변경되는 개념이 아니라 인덱스를 삭제하고 다시 추가하는 원리로 작동을 한다. 이 2가지 동작을 연속으로 하면 다른 DML의 문장들보다 더 큰 부하를 주게 된다.

2. 타 SQL 실행에 악영향을 준다.
- 테이블의 수행시간이 0.1초로 가정하면, 테이블에 인덱스를 박아두게 된다면 쿼리 수행시간이 느려지는 경우가 발생을 한다. 그래서 새로 생성하는 인덱스에 대해서는 다른 SQL 문장에 대해서 고려를 하고 실행하길 바란다.

8. 인덱스의 조회
select table_name, index_name
from user_indexes
where table_name='DEPARTMENTS';
이를 작성하게 된다면 여태껏 형성된 인덱스들에 대해서 출력을 하게 된다. 더욱 더 놀라운 사실은 기본키로 지정된 인덱스에 대해서는 자동적으로 인덱스가 추가되는 개념이다. 이름도 제약조건이랑 똑같이 추가되니깐 알아두면 약이다.

9. 인덱스 삭제하기(인덱스 중단에 대해서는 굳이 생략하겠습니다...)
Drop Index DEPT_ID_PK;
사용하지 않는 인덱스는 삭제하길 권장하지만, 무엇보다 인덱스를 삭제하기 전에는 신중하게 생각해야 하는 점을 알아두도록 하자.

(참조) 사용자로부터 입력을 받게끔 하기
C언어에서 Scanf를 배웠을 것이다. 그래서 다음과 같이 변수 앞에 &를 써두면 사용자로부터 입력한 값들로서 탐색을 도와주고, 이 기능은 오라클SQL 이외엔 없다고 생각하면 되니깐 알아두면 좋겠다.
undefine SALARY
Select department_id, last_name
from employees
where salary Between &&SALARY and &&SALARY
본래 &만 써주면 각 입력해야하는 부분에 있어서 계속 입력을 받게 되지만, &&를 써주면 한 값에 대해서 계속 적용해 나아간다는 의미로 생각하면 되고, 이에 대해서 수정을 하려면 맨 윗줄에 Undefine 변수이름을 써주길 바란다. 참고로 숫자는 이렇게 하고, 문자열에 대해서는 '&&LAST_NAME'으로 설정하면 된다.

---

여태껏 정리한 내용을 구독해 주셔서 감사드립니다. 중간고사에 좋은 성과가 있으시길 바랍니다. 질문 내용이나 잘못된 내용, SQL 관련해서 질문할 내용은 13학번 강인성에게 연락주시면 답변 드리겠습니다.