데이터 독립성이 보장된 최초의 데이터 모델이다.

물리(Physical) = Ordering, Indexing, Access Path 종속적

논리(Logical) = 데이터 항목 자체에 집중

테이블 형태, 중복이 최소화 된 데이터 모델이다.

데이터 모델을 봤을 때 이해하는 법

DVD Rental Model 로 보자

Reference Data(사업 환경 설정) : 언어, 카테고리, 나라, 도시 등

Master Data(사업 활동 주체와 대상) : 고객, 상품, 가게, 영화, 고객 주소 등

Transaction Data(사업 활동) : 돈과 관련, 주문 등

재사용률은 위에서 부터 아래 순이다.

**SELECT 문법**

SELECT column FROM table\_name;

모든 데이터를 다 보고 싶다면 column 에 \* 넣으면 된다.

From - >table\_name -> select -> column 순으로 실행된다.

**Alias : 코드의 가독성을 높여준다, SQL 성능을 높여준다**

DBMS -> 옵티마이저(최적화 기능 담당)이 SQL을 읽을 때 실행할 때 가장 빠르고 저비용인 방법을 찾는다.

테이블이 여러개 있을 때 table\_name 뒤에 alias를 붙여주는 걸 의미한다.

SELECT a.column FROM table\_name a;

**ORDER BY 문법 : 데이터를 정렬하는데 사용. 업무 처리상 매우 중요**

SELECT column\_1, column\_2 FROM table\_name ORDER BY column\_1 ASC(오름차순), column\_2 DESC(내림차순)

default값은 ASC이다.

Asc + desc 정렬 : first\_name을 ASC로, last\_name을 DESC로 정렬 가능

ORDER BY 에 정수 넣어도 된다. 정수가 의미하는건 COLUMN을 왼쪽으로 부터 오른쪽으로 읽었을 때 순서이다. 간단하지만 가독성이 좋은 방법은 아니다. (추천X)

ORDER BY COLUMN\_1, COLUMN\_2 : COLUMN\_1을 정렬하고 그걸 기반으로 COLUMN\_2를 정렬한다.

**SELECT DISTINC 문** : **중복 값 제외한 결과값이 출력된다.** 즉, 같은 결과의 행이면 중복을 제거할 수 있다.

SELECT

*//COLUMN\_1 + COLUMN\_2 중복값을 제거한다. 따라서 COLUMN\_1에는 중복값 있을 수 있다.*

DISTINCT COLUMN\_1, COLUMN2

FROM TABLE\_NAME

ORDER BY COLUMN\_1, COLUMN\_2;

참고) create table은 ddl이기에 commit 필요없지만, insert문은 필요하다.

컬럼 한 개 : 한 개의 컬럼의 중복값을 모두 제거한다.

컬럼 두 개 이상 : 모든 컬럼을 합친 ROW 값의 중복값을 제거한다.

**SELECT DISTINCT ON (COLUMN1) COLUMN\_1, COLUMN\_2 : COLUMN\_1 값 기준 중복 제거. COLUMN\_2값은 단 한 개 값 만을 보여줌**

****

****

**Bcolor의 모든 중복값들을 제거하고, 오른편의 fcolor column에는 오름차순 맨 위에 있는 값 단 한 개 만을 둔다.**

**SELECT DISTINCT ON (COLUMN1) COLUMN\_1, COLUMN\_2 … ORDER BY COLUMN\_1, COLUMN\_2 DESC;**

**: FCOLOR 컬럼 값 보여줄 때 내림차순 정렬함.**

****

**WHERE 절 : 집합을 가져올 때 어떤 집합을 가져올 것인가에 대한 조건을 설정하는 절이다.**

SELECT

COLUMN\_1, COLUMN\_2

FROM

TABLE\_NAME

WHERE

<조건> //문자열을 조건으로 작성 시 작은 따옴표 써야함!

;

연산자



조건이 두 개 이상일 때는 AND를 붙인다.

참고) CTRL + SHIFT + F 누르면 정렬된다.

**LIMIT 절 :** **특정 집합을 출력 시 출력하는 행의 수를 한정하는 역할을 한다. 부분 범우 처리 시 사용된다. PostgreSQL, MySQL등에서 지원한다.**

SELECT \*

FROM TABLE\_NAME

LIMIT N OFFSET M //출력하는 행의 수를 지정하한다. 시작 OFFSET M값의 시작위치는 0이다.

;

가져오는 데이터 양이 방대할 때, 출력 행의 수를 제한하여 보여준다.

예를 들어 구글 플레이에서 앱을 보여줄 때 1억건이 있다고 해서 1억건을 다 보여주지는 않는다.

따라서 제한하여 보여준다. LIMIT 5 하면 행 다섯개만 보여준다. OFFSET 3 하면 0부터 시작이기에 4번째 행부터 보여준다(INDEX로 생각하면 쉽다).

**FETCH 절 : 특정 집합을 출력 시 출력하는 행의 수를 한정하는 역할을 한다. 부분 범우 처리 시 사용된다. (매우 중요!)**

SELECT \*

FROM TABLE\_NAME

OFFSET N ROWS //출력하는 행의 수를 지정하면서 시작 위치를 지정한다. (옵션)

FETCH FIRST [N] ROW ONLY // FIRST N개의 데이터만 가져오겠다(ROW ONLY).

;

OFFSET으로 첫 시작 출력 행을 지정한다.

한 건만 조회하려면 n에 아무것도 넣지 않으면 된다. (선입선출 방식에 많이 쓰인다)

N에 넣는 정수 크기만큼의 행이 출력된다.

**IN 연산자** : 특정 집합(컬럼 혹은 리스트)에서 특정 집합 혹은 리스트가 존재하는지 판단하는 연산자이다.

SELECT \*

FROM TABLE\_NAME

WHERE COLUMN\_NAME IN (VALUE1, VALUE2…) //COLUMN\_NAME이 가지고 있는 집합에서 VALUE1,2 등의 값이 존재하는지 확인

SELECT \*

FROM TABLE\_NAME

WHERE COLUMN\_NAME IN

(SELECT COLUMN\_NAME2 FROM TABLE\_NAME2) // COLUMN\_NAME이 가지고 있는 집합에서 TABLE\_NAME2 테이블의 COLUMN\_NAME2의 집합이 존재하는지 확인

EX)

**select**

CUSTOMER\_ID,

RENTAL\_ID,

RETURN\_DATE

**from**

rental

**where**

CUSTOMER\_ID **in** (1, 2) //CUSTOMER\_ID 가 1 혹은 2 인것

**order** **by**

RETURN\_DATE **desc**;

다음과 같다.

WHERE CUSTOMER\_ID = 1

OR CUSTOMER\_ID =2

IN연산자를 사용하면 가독성이 좋다.

옵티마이저의 특성상(DBMS 최적화기, SQL최적화) IN 조건 성능상 유리

**select**

CUSTOMER\_ID,

RENTAL\_ID,

RETURN\_DATE

**from**

rental

**where**

CUSTOMER\_ID **not** **in** (1, 2) //CUSTOMER\_ID가 1 OR 2가 아닌것(1,2 제외 나머지)

**order** **by**

RETURN\_DATE **desc**;

NOT IN 연산자는 CUSTOMER\_ID <> 1 AND CUSTOMER\_ID <> 2와 같다.

가능한 NOT IN을 사용하는게 가독성 좋다.

IN 연산자 실습 – 서브쿼리

**select**

FIRST\_NAME, //RETURN\_DATE가 2015년 5월 27일인 CUSTOMER\_ID의 FIRST\_NAME,LAST\_NAME 출력

LAST\_NAME

**from**

customer

**where**

customer\_id **in** (

**select**

customer\_id

**from**

rental

**cast**(RETURN\_DATE **as** **DATE**) = '2015-05-27' //RETURN\_DATE가 '2015-05-27'인 CUSTOMER\_ID 출력

)

**BETWEEN 연산자 : 특정 집합에서 어떠한 컬럼의 값이 특정 범위 안에 들어가는 집합을 출력하는 연산자이다.**

SELECT \*

FROM TABLE\_NAME

WHERE COLUMN\_NAME

BETWEEN VALUE\_A AND VALUE\_B; //COLUMN\_NAME 컬럼의 값이 VALUE\_A와 VALUE\_B 사이에 있는 집합 출력.

COLUMN\_NAME >= VALUE\_A AND COLUMN\_NAME <= VALUE\_B 와 같다.

**NOT BETWEEN은 COLUMN\_NAME < VALUE\_A AND COLUMN\_NAME > VALUE\_B와 같다.**

**select**

CUSTOMER\_ID,

PAYMENT\_ID,

AMOUNT

**from**

payment

**where**

AMOUNT **between** 8 **and** 9;

AMOUNT NOT BETWEEN 8 AND 9 == AMOUNT < 8 OR AMOUNT > 9

**select**

CUSTOMER\_ID,

PAYMENT\_ID,

AMOUNT,

PAYMENT\_DATE

**from**

payment

**where**

**cast**(PAYMENT\_DATE **as** **DATE**) **between** '2007-02-07' **and** '2007-02-15';

CAST(PAYMENT\_DATE AS DATE) == TO\_CHAR(PAYMENT\_DATE, ‘YYYY-MM-DD’)

**LIKE 연산자 : 특정 집합에서 어떠한 컬럼의 값이 특정 값과 유사한 패턴을 갖는 집합을 출력하는 연산자이다.**

SELECT \*

FROM TABLE\_NAME

WHERE COLUMN\_NAME

LIKE 특정패턴 // COLUMN\_NAME의 값이 특정 패턴과 유사한 집합을 출력한다.

NOT LIKE 특정패턴 // COLUMN\_NAME의 값이 특정 패턴과 유사하지 않는 집합을 출력한다.

특정 패턴에서

%는 어떤 문자 혹은 문자열이라든지 매칭 되었다고 판단한다.

\_는 한 개의 문자가 어떤 문자이든지 매칭 되었다고 판단한다.

**select**

FIRST\_NAME,

LAST\_NAME

**from**

customer

**where**

FIRST\_NAME **like** 'Jen%' //출력 결과 : Jennifer, Jennie etc.

;

LIKE 연산자 실습

SELECT

‘FOO’ LIKE ‘FOO’ , //결과값 TRUE

‘FOO’ LIKE ‘F%’, //TRUE

‘FOO LIKE ‘\_O\_’, //TRUE

‘BAR’ LIKE ‘B\_’ //FALSE

;

**select**

FIRST\_NAME,

LAST\_NAME

**from**

customer

**where**

FIRST\_NAME **like** '%er%' //FIRST\_NAME 이 er을 포함하기만 하면 됨.

;

**select**

FIRST\_NAME,

LAST\_NAME

**from**

customer

**where**

FIRST\_NAME **like** '\_her%' //첫 번째 문자에 관계없이 바로 her이 연결되는 문자 매칭.

;

**IS NULL 연산자 : 특정 컬럼 혹은 값이 널 값인지 아닌지 판단.**

SELECT \*

FROM TABLE\_NAME

WHERE COLUMN\_NAME

IS NULL // COLUMN\_NAME이 NULL인 집합을 출력한다.

IS NOT NULL // COLUMN\_NAME이 NOT NULL인 집합을 출력한다.

**주의!**

NULL은 = 연산자 사용불가. 즉 COLUMN\_NAME = NULL 이런식으로 불가!!

참고) INSERT, UPDATE, DELETE 하기 위해서는 COMMIT 혹은 ROLLBACK을 해야한다. 이를 트랜잭션 처리라고 한다.

**조인이란? : 두 개 이상의 테이블에 있는 정보 중 사용자가 필요한 집합에 맞게 가상의 테이블처럼 만들어서 결과를 보여주는 것이다.**

**INNER 조인 : 특정 컬럼 기준으로 정확히 매칭된 집합을 출력한다. 대표 조인이다.**

****

SELECT A.ID ID\_A

, A.FRUIT FRUIT\_A

, B.ID ID\_B

, B.FRUIT FRUIT B

FROM BASKET\_A A //FROM BASKET\_A A, BASKET\_B B WHERE A.FRUIT = B.FRUIT으로 나타낼수도 있다.

INNER JOIN BASKET\_B B

ON A.FRUIT = B.FRUIT //BASKET\_A/B를 FRUIT 컬럼 기준으로 조인한다.

SELECT

A.CUSTOMER\_ID, A.FIRST\_NAME

, A.LAST\_NAME, A.EMAIL

, B.AMOUNT, B.PAYMENT\_DATE

FROM CUSTOMER A

INNER JOIN PAYMENT B

ON A.CUSTOMER\_ID = B.CUSTOMER\_ID

WHERE A.CUSTOMER\_ID = 2;

고객은 여러건의 결제를 할 수 있다. 고객 1: 결제M 관계!

세 개의 테이블 조인

SELECT

A.CUSTOMER\_ID, A.FIRST\_NAME

, A.LAST\_NAME, A.EMAIL

, B.AMOUNT, B.PAYMENT\_DATE

, C.FIRST\_NAME AS S\_FIRST\_NAME

, C.LAST\_NAME AS S\_LAST\_NAME

FROM CUSTOMER A

INNER JOIN PAYMENT B

ON A.CUSTOMER\_ID = B.CUSTOMER\_ID

INNER JOIN STAFF C

ON B.STAFF\_ID = C.STAFF\_ID

고객1 : 결제 M : 직원 1

**OUTER 조인 : 특정 컬럼 기준으로 매칭된 집합을 출력하지만 한쪽의 집합은 모두 출력하고 다른 한쪽의 집합은 매칭되는 컬럼의 값 만을 출력한다.**

****

**<LEFT OUTER 조인> 기준이 되는 테이블은 FROM절에 나온다.**

SELECT

A.ID AS ID\_A

, A.FRUIT AS FRUIT\_A

, B.ID AS ID\_B

, B.FRUIT AS FRUIT\_B

FROM

*BASKET\_A A*

**LEFT JOIN** BASKET\_B B

ON A.FRUIT = B.FRUIT



**<LEFT OUTER 조인 – LEFT ONLY> - WHERE 절 빼먹지말자!!!!**

****

SELECT

A.ID AS ID\_A

, A.FRUIT AS FRUIT\_A

, B.ID AS ID\_B

, B.FRUIT AS FRUIT\_B

FROM

BASKET\_A A

**LEFT JOIN** BASKET\_B B

ON A.FRUIT = B.FRUIT

**WHERE B.ID IS NULL**; //ID가 없는 FRUIT은 없다! 즉 A집합만 가져온 다.

**<RIGHT OUTER 조인> 기준이 되는 테이블은 RIGHT JOIN 다음에 나온다.**

****

SELECT

A.ID AS ID\_A

, A.FRUIT AS FRUIT\_A

, B.ID AS ID\_B

, B.FRUIT AS FRUIT\_B

FROM

BASKET\_A A

**RIGHT JOIN** BASKET\_B B

ON A.FRUIT = B.FRUIT

**<RIGHT OUTER 조인 – RIGHT ONLY>**

****

SELECT

A.ID AS ID\_A

, A.FRUIT AS FRUIT\_A

, B.ID AS ID\_B

, B.FRUIT AS FRUIT\_B

FROM

BASKET\_A A

**RIGHT JOIN** BASKET\_B B

ON A.FRUIT = B.FRUIT

WHERE A.ID IS NULL;

**SELF조인 : 동일한 테이블 끼리의 특정 컬럼을 기준으로 매칭되는 집합을 출력한다.**

**동일한 테이블이지만 각각의 다른 집합으로 구성해놓고 조인한다.**

****

SELECT

E.FIRST\_NAME || ‘ ‘ || E.LAST\_NAME EMPLOYEE

, M.FIRST\_NAME || ‘ ‘ || M.LAST\_NAME MANAGER

FROM

EMPLOYEE E

INNER JOIN EMPLOYEE M

ON M.EMPLOYEE\_ID = E.MANAGER\_ID

ORDER BY MANAGER;



각 직원의 상위 관리자들이 나온다.

**<SELF 조인 - SELF LEFT OUTER >**

SELECT

E.FIRST\_NAME || ‘ ‘ || E.LAST\_NAME EMPLOYEE

, M.FIRST\_NAME || ‘ ‘ || M.LAST\_NAME MANAGER

FROM

EMPLOYEE E

LEFT JOIN EMPLOYEE M

ON M.EMPLOYEE\_ID = E.MANAGER\_ID

ORDER BY MANAGER;

****

각 직원의 상위 관리자를 출력하면서 모든 직원을 출력함.

최고 관리자인 ‘Windy Hays’가 결과 집합에 출력됨

SELECT

F1.TITLE

, F2.TITLE

, F1.LENGTH

FROM

FILM F1

INNER JOIN FILM F2

ON F1.FILM\_ID <> F2.FILM\_ID

AND F1.LENGTH = F2.LENGTH; //상영 시간은 동일한데 다른 영화제목

**FULL OUTER조인 : INNER, LEFT OUTER, RIGHT OUTER 조인 집합을 모두 출력한다.**

****

SELECT

A.ID ID\_A

**,** A.FRUIT FRUIT\_A

, B.ID ID\_B

, B.FRUIT FRUIT

FROM BASKET\_A A

FULL OUTER JOIN BASKET\_B B

ON A.FRUIT = B.FRUIT;



SELECT

E.EMPLOYEE\_NAME

, D.DEPARTMENT\_NAME

FROM EMPLOYEES E

FULL OUTER JOIN DEPARTMENTS D

ON.D.DEPARTMENT\_ID = E.DEPARTMENT\_ID;



**<FULL OUTER 조인 – ONLY OUTER>**

SELECT

A.ID ID\_A

**,** A.FRUIT FRUIT\_A

, B.ID ID\_B

, B.FRUIT FRUIT

FROM BASKET\_A A

FULL OUTER JOIN BASKET\_B B

ON A.FRUIT = B.FRUIT;

WHERE A.ID IS NULL

OR B.ID IS NULL



**<FULL OUTER조인 – RIGHT ONLY> == RIGHT OUTER – RIGHT ONLY (FULL OUTER JOIN을 RIGHT JOIN으로 변경하여도 값이 똑같다.**

SELECT

E.EMPLOYEE\_NAME

, D.DEPARTMENT\_NAME

FROM EMPLOYEES E

FULL OUTER JOIN DEPARTMENTS D

ON.D.DEPARTMENT\_ID = E.DEPARTMENT\_ID;

WHERE E.EMPLOYEE\_NAME IS NULL;

**CROSS 조인 : Cartesian Product이라고도 하며 조인되는 두 테이블에서 곱집합으로 반환한다.**

****

SELECT \*

FROM CROSS\_T1

CROSS JOIN CROSS\_T2;

****

위의 SQL문은 아래와 같다.

SELECT \*

FROM CROSS\_T1, CROSS\_T2;

**NATURAL 조인 : 특정 테이블의 같은 이름을 가진 컬럼 간의 조인집합을 출력한다. INNER 조인 집합 결과를 출력한다. SQL문 자체가 간소해 지는 방법이다.**

****

한 개의 카테고리는 여러개의 프로덕트를 가질 수 있다.

SELECT \*

FROM PRODUCTS A

NATURAL JOIN CATEGORIES B;

NATURAL 조인은 INNER 조인의 또 다른 SQL 작성 방식이다. 즉, 조인 컬럼을 명시하지 않아도 된다.

특징은 ON이 없으며 이유는 자동으로 매칭해주기 때문이다 -> 안정성 문제 때문에 아직 잘 사용은 안한다.



위의 코드는 아래와 같다.

SELECT

A.CATEGORY\_ID, A.PRODUCT\_ID

,A.PRODUCT\_NAME, B.CATEGORY\_NAME

FROM PRODUCT A

INNER JOIN CATEGORIES B

ON A.CATEGORY\_ID = B.CATEGORY\_ID;

혹은

SELECT

A.CATEGORY\_ID, A.PRODUCT\_ID

,A.PRODUCT\_NAME, B.CATEGORY\_NAME

FROM PRODUCT A, CATEGORIES B

WHERE A.CATEGORY\_ID = B.CATEGORY\_ID;

**GROUP BY 절 : SELECT 문에서 반환된 행을 그룹으로 나눈다. 각 그룹에 대한 합계, 평균, 카운트 등을 계산할 수 있다.**

**MySQL에서 유형별로 갯수를 가져오고 싶은데, 단순히 COUNT 함수로 데이터를 조회하면 전체 갯수만을 가져옵니다.이렇게 유형별로 갯수를 알고 싶을 때는 컬럼에 데이터를 그룹화 할 수 있는 GROUP BY를 사용하는 것입니다.**

****

SELECT

COLUMN\_1,

집계합수(COLUMN\_2)

FROM TABLE\_NAME

GROUP BY COLUMN\_1; //GROUP BY 절은 FROM 또는 WHERE 절 바로 뒤에 나타나야한다.

SELECT CUSTOMER\_ID

FROM PAYMENT

GROUP BY CUSTOMER\_ID; //중복 값이 제거된 CUSTOMER\_ID를 구할 수 있다.

위의 코드는 아래와 동일하다.

SELECT DISTINCT CUSTOMER\_ID

FROM PAYMENT;

**select**

customer\_id,

**sum**(amount) **as** amount\_sum

**from**

payment

**group** **by**

customer\_id

**order** **by**

**sum**(amount) **desc**; //sum(amount)는 amount\_sum 혹은 2번으로 변경가능하다.

**select**

staff\_id,

**count**(payment\_id) **as** **count**

**from**

payment

**group** **by**

staff\_id; //각 staff가 몇건의 payment를 처리했는가

집계 함수를 쓰지 않은 컬럼은 GROUP BY 절에 있어야한다.

**HAVING 절 : GROUP BY 절과 함께 사용하여 GROUP BY 결과를 특정 조건으로 필터링 하는 기능을 한다.**

SELECT COLUMN\_1

,집계함수(COLUMN\_2)

FROM TABLE\_NAME

GROUP BY COLUMN\_2

HAVING 조건식;

HAVING 절은 GROUP BY 절에 의해 생성된 그룹행의 조건을 설정한다.

반면, WHERE 절은 GROUP BY 절이 적용되기 전에 개별 행의 조건을 설정한다.

**select**

customer\_id,

**sum**(amount) **as** amount\_sum

**from**

payment

**group** **by**

customer\_id

**having** **sum**(amount)>200;

**UNION 연산 : 두 개 이상의 SELECT문들의 결과 집합을 단일 결과 집합으로 결합하며 결합시 중복된 데이터는 제거된다. 두 개의 SELECT문 간 컬럼의 개수는 동일해야 하고 해당 순서의 열에는 서로 호환되는 데이터 유형이어야 한다. ORDER BY로 정렬하고자 할 경우 마지막 SELECT 문에 사용해야한다.**

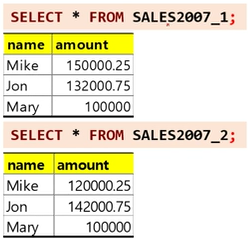
SELECT COLUMN\_1\_1, COLUMN\_1\_2

FROM TABLE\_NAME\_1

UNION

SELECT COLUMN\_2\_1, COLUMN\_2\_2

FROM TABLE\_NAME\_2;



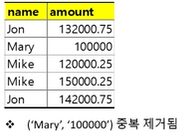
SELECT \*

FROM SALES2007\_1

UNION

SELECT \*

FROM SALES2007\_2



SELECT NAME

FROM SALES2007\_1

UNION

SELECT NAME

FROM SALES2007\_2



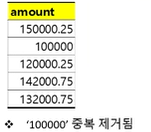
SELECT AMOUNT

FROM SALES2007\_1

UNION

SELECT AMOUNT

FROM SALES2007\_2



**UNION ALL 연산 : 두 개 이상의 SELECT문들의 결과 집합을 단일 결과 집합으로 결합하며 결합 시 중복된 데이터도 모두 출력한다.**

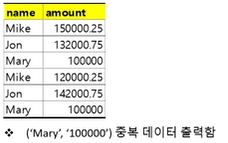
SELECT \*

FROM SALES2007\_1

UNION ALL

SELECT \*

FROM SALES2007\_2



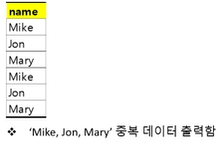
SELECT NAME

FROM SALES2007\_1

UNION

SELECT NAME

FROM SALES2007\_2



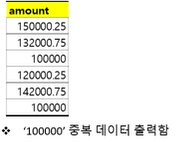
SELECT AMOUNT

FROM SALES2007\_1

UNION

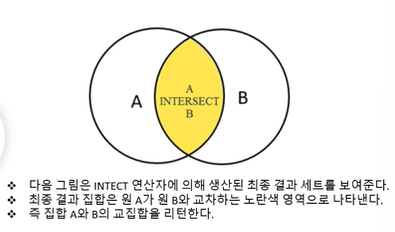
SELECT AMOUNT

FROM SALES2007\_2



**INTERSECT 연산 : 두 개 이상의 SELECT문들의 결과 집합을 하나의 결과 집합으로 결합한다.**

**ORDER BY는 마지막 SELECT문에 사용**

****

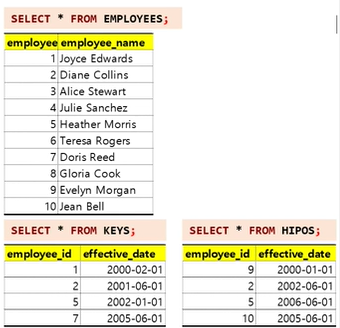
SELECT COLUMN\_1\_1, COLUMN\_1\_2

FROM TABLE\_NAME\_1

INTERSECT

SELECT COLUMN\_2\_1, COLUMN\_2\_2

FROM TABLE\_NAME\_2



SELECT EMPLOYEE\_ID

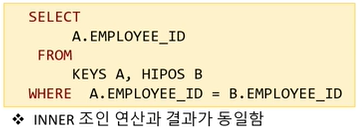
FROM KEYS

INTERSECT

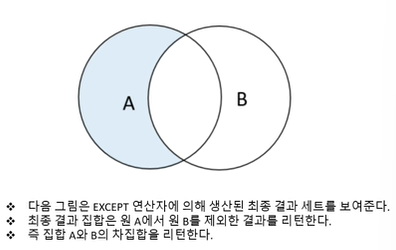
SELECT EMPLOYEE\_ID

FROM HIPOS;





**EXCEPT 연산 : 맨 위의 SELECT문의 결과 집합에서 그 아래에 있는 SELECT문의 결과 집합을 제외한 결과를 리턴한다. ORDER BY는 맨 마지막 SELECT문에 ORDER BY적을 사용한다. (실무에서 많이 사용된다.)**



SELECT COLUMN\_1\_1, COLUMN\_1\_2

FROM TABLE\_NAME\_1

EXCEPT

SELECT COLUMN\_2\_1, COLUMN\_2\_2

FROM TABLE\_NAME\_2

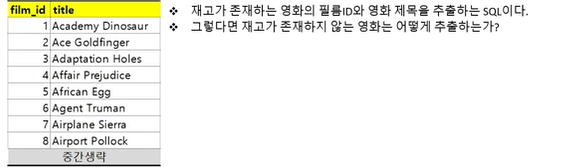
SELECT DISTINCT INVENTORY.FILM\_ID, FILM.TITLE

FROM INVENTORY

INNER JOIN FILM

ON FILM.FILM\_ID = INVENTORY\_FILM\_ID

ORDER BY TITLE;



온전한 테이블 하나에 EXCEPT문 붙여 재고 있는 테이블을 뺀다.

SELECT FILM\_ID, TITLE

FROM FILM //모든 영화

EXCEPT

SELECT DISTINCT INVENTORY.FILM\_ID, FILM.TITLE //재고 있는 영화

FROM INVENTORY

INNER JOIN FILM

ON FILM.FILM\_ID = INVENTORY\_FILM\_ID

ORDER BY TITLE;



**서브쿼리 : SQL문 내에서 메인 쿼리가 아닌 하위에 존재하는 쿼리를 말한다. 두 개의 SQL을 결합하여 하나의 SQL문으로 결과를 도출하려고 만들었다. (중첩 서브쿼리, 인라인 뷰, 스칼라 서브쿼리)**

**중첩 서브쿼리 (WHERE 절)**

SELECT FILM\_ID, TITLE, RENTAL\_RATE

FROM FILM

WHERE RENTAL\_RATE > (

SELECT AVG(RENTAL\_RATE)

FROM FILM

);

**인라인 뷰 서브쿼리(FROM 절)**

SELECT A.FILM\_ID, A.TITLE, A.RENTAL\_RATE

FROM FILM A

, (

SELECT AVG(RENTAL\_RATE) AS AVG\_RENTAL\_RATE

FROM FILM

) B

WHERE A.RENTAL\_RATE > B.AVG\_RENTAL\_RATE;

**스칼라 서브쿼리(SELECT 문)**

SELECT A.FILM\_ID, A.TITLE, A.RENTAL\_RATE

FROM (

SELECT A.FILM\_ID, A.TITLE, A.RENTAL\_RATE, (

SELECT AVG(L.RENTAL\_RATE)

FROM FILM L

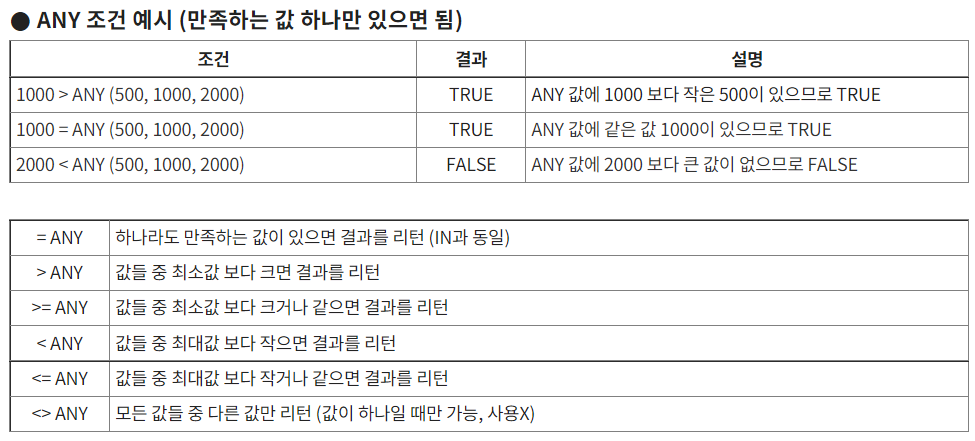
) AS AVG\_RENTAL\_RATE

FROM FILM A

) A

WHERE A.RENTAL\_RATE > A.AVG.RENTAL\_RATE;

**ANY 연산자 : 서브쿼리의 값이 어떠한 값(하나만)이라도 만족을 하면 조건이 성립된다.**

****

**>=ANY**

SELECT TITLE, LENGTH //영화 분류별 상영시간이 가장 긴 영화의 제목 및 상영시간을 구함

FROM FILM

WHERE LENGTH >= **ANY** (

SELECT MAX(LENGTH) //영화 분류멸 상영시간이 가장 긴 상영시간을 구함

FROM FILM A, FILM\_CATEGORY B

WHERE A.FILM\_ID = B.FILM\_ID

GROUP BY B.CATEGORY\_ID );

**=ANY(‘IN’과 동일)**

SELECT TITLE, LENGTH

FROM FILM

WHERE LENGTH = **ANY** (

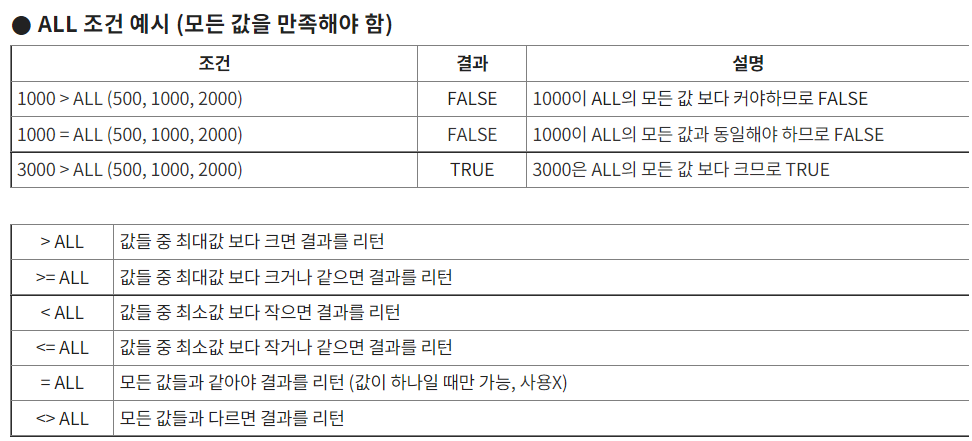
SELECT MAX(LENGTH)

FROM FILM A, FILM\_CATEGORY B

WHERE A.FILM\_ID = B.FILM\_ID

GROUP BY B.CATEGORY\_ID );

**ALL 연산자 : 서브쿼리의 모든 값이 만족을 해야만 조건 성립**

****

SELECT TITLE, LENGTH

FROM FILM

WHERE LENGTH >= ALL (

SELECT MAX(LENGTH)

FROM FILM A, FILM\_CATEGORY B

WHERE A.FILM\_ID = B.FILM\_ID

GROUP BY B.CATEGORY\_ID

);

SELECT FILM\_ID, TITLE, LENGTH // 평가 기준 평균값들보다 상영시간이 긴 영화정보 출력

FROM FILM

WHERE LENGTH > ALL (

SELECT ROUNG(AVG(LENGTH), 2) //평가 기준으로 영화 상영 길이의 평균 값

FROM FILM

GROUP BY RATING)

ORDER BY LENGTH;

<https://gent.tistory.com/287>

**EXISTS 연산자 : 서브쿼리 내에 집합이 존재하는지 존재 여부만 판단한다. 존재 여부만을 판단하므로 연산 시 부하가 줄어든다. IN절과 동등 (많이 사용됨!!!) (WHERE절)**

**해당 집합이 존재하기만 하면 더이상 연산을 하지 않으므로 성능상 유리**

SELECT FIRST\_NAME, LAST\_NAME

FROM CUSTOMER C

WHERE

EXISTS (

SELECT 1 //해당 조건을 만족하면 1을 반환한다. TRUE(1)면 메인쿼리를 실행한다.

FROM PAYMENT P

WHERE P.CUSTOMER\_ID = C.CUSTOMER\_ID //메인 테이블과 연결

AND P.AMOUNT > 11 )

ORDER BY FIRST\_NAME, LAST\_NAME;

**NOT EXISTS 연산자 (NOT IN과 차이점 : NOT EXISTS는 NULL 데이터도 포함!!)**

SELECT FIRST\_NAME, LAST\_NAME

FROM CUSTOMER C

WHERE

NOT EXISTS (

SELECT 1

FROM PAYMENT P

WHERE P.CUSTOMER\_ID = C.CUSTOMER\_ID

AND P.AMOUNT > 11 )

ORDER BY FIRST\_NAME, LAST\_NAME;

**OVER 절 : 단위함수랑 집계함수를 같이 사용할 수 있게 만들어주는 절. Subquery와 Group by를 없애준다.**

예) 직원연봉, 직원 전체연봉, 직원 전체연봉 총합 테이블작성

SELECT EMPLOYEE\_ID, SALARY,

ROUND(( SELECT AVG(SALARY) FROM EMPLOYEES)) AS 직원평균연봉,

(SELECT SUM(SALARY) FROM EMPLOYEES) AS 전체연봉

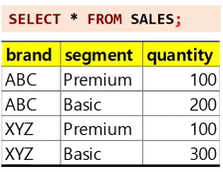
FROM EMPLOYEES

* OVER 함수 사용

SELECT EMPLOYEE\_ID, SALARY, ROUND(AVG(SALARY) OVER()) 직원평균연봉

SUM(SALARY) OVER() 전체연봉 FROM EMPLOYEES

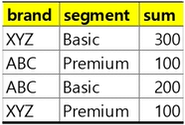
**GROUPING SET 절 : 여러 개의 UNION ALL을 이용한 SQL과 같은 결과를 도출할 수 있다.**

****

SELECT BRAND, SEGMENT, SUM(QUANTITY)

FROM SALES

GROUP BY BRAND, SEGMENT;



SELECT BRAND,SUM(QUANTITY)

FROM SALES

GROUP BY BRAND



SELECT SEGMENT, SUM(QUANTITY)

FROM SALES

GROUP BY SEGMENT;



SELECT SUM(QUANTITY)

FROM SALES;



위의 SELECT문들을 UNION ALL 하면 아래와 같다



단점 : 동일한 데이터 4번 읽음 -> 성능저하,

SQL문이 너무 길다 -> 유지보수 용이 X

GROUPING SETS 절을 이용하면 한번에 다양한 기준의 컬럼 조합으로 집계를 구할 수 있다.

SELECT C1, C2, 집계함수(C3)

FROM TABLE\_NAME

GROUP BY

GROUPING SETS (

(C1, C2), (C1), (C2), () );

SELECT BRAND, SEGMENT, SUM(QUANTITY)

FROM SALES

GROUP BY

GROUPING SETS (

(BRAND, SEGMENT),

(BRAND),

(SEGMENT),

()

);

**<GROUPING SET – GROUPING 함수> : 해당 컬럼이 집계에 사용되었으면 0, 아니면 1 반환**

SELECT GROUPING(BRAND) GROUPING\_BRAND

, GRUOPING(SEGMENT) GROUPING\_SEGMENT

, BRAND

, SEGMENT

, SUM(QUANTITY)

FROM SALES

GROUP BY

GROUPING SETS (

(BRAND, SEGMENT)

,(BRAND)

,(SEGMENT)

,()

)

ORDER BY BRAND, SEGMENT;



**ROLL UP : 지정된 GROUPING 컬럼의 소계를 생성하는데 사용된다.**

SELECT C1, C2, C3, 집계함수(C4)

FROM TABLE\_NAME

GROUP BY

ROLL UP (C1, C2, C3) //소계를 생성할 컬럼을 지정한다. 컬럼 지정 순서에 따라 결과값이 달라질 수 있다. (제일 앞에 놓인 것에 소계를 구함)

SELECT C1, C2, C3, 집계함수(C4)

FROM TABLE\_NAME

GROUP BY C1

ROLL UP (C2, C3) //특정 컬럼을 제외한 ROLL UP 도 가능

SELECT BRAND, SEGMENT, SUM(QUANTITY)

FROM SALES

GROUP BY

ROLLUP (BRAND, SEGMENT)

ORDER BY BRAND, SEGMENT;



**부분 ROLLUP : GROUP BY별 합계 + 맨 앞에 쓴 컬럼별 합계**

SELECT SEGMENT, BRAND, SUM(QUANTITY)

FROM SALES

GROUP BY SEGMENT,

ROLLUP (BRAND)

OLRDER BY SEGMENT, BRAND;



**CUBE절 : 지정된 GROUPING 컬럼의 다차원 소계를 생성하는데 사용된다.**

SELECT C1, C2, C3

FROM TABLE\_NAME

GROUP BY

CUBE (C1, C2, C3)

SELECT C1, C2, C3, 집계함수(C4)

FROM TABLE\_NAME

GROUP BY C1,

CUBE(C2, C3);

CUBE(C1,C2,C3)

* GROUPING SETS (

(C1, C2, C3),

(C1, C2),

(C1, C3),

(C2, C3),

(C1),

(C2),

(C3),

()

)

CUBE(C1,C2,C3) 를 GROUPING SETS로 표현하면 총 8개의 소계가 발생한다.

CUBE절 내 인자의 개수가 3개이면 2의 3승의 소계가 발생한다 (2^N = 소계의 수)

SELECT BRAND,SEGMENT,SUM(QUANTITY)

FROM SALES

GROUP BY

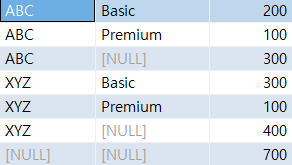
CUBE(BRAND, SEGMENT) //ROLLUP과 차이 : ROLLUP은 SEGMENT별 합계 제공 X

ORDER BY BRAND, SEGMENT;

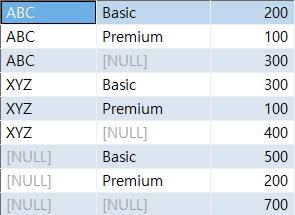


총 4개의 경우의 수가 합계로 출력되었다.

ROLLUP 사용시



CUBE 사용시



SEGMENT별 소계를 확인할 수 있다.

ROLLUP (a, b, c) : (a, b, c) / (a, b) / (a) / ()

CUBE (a, b, c) : (a, b, c) / (a, b) / (a, c) / (b, c) / (a) / (b) / (c) / ()

**부분 CUBE**

SELECT BRAND, SEGMENT, SUM(QUANTITY)

FROM SALES

GROUP BY BRAND, CUBE(SEGMENT)

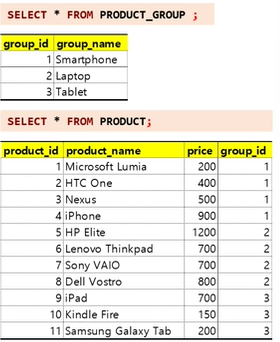
ORDER BY BRAND, SEGMENT;



SEGMENT별 소계 X

전체 테이블 기준으로 소계하지 않는다.

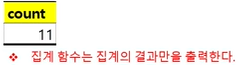
**분석 함수 : 특정 집합 내에서 결과 건수의 변화 없이 해당 집합 안에서 합계 및 카운트 등을 계산할 수 있는 함수이다.**



**<분석함수 OVER()>**

SELECT COUNT(\*)

FROM PRODUCT;



SELECT COUNT(\*) OVER(), A.\*

FROM PRODUCT A



**<분석함수 AVG()>**

****

SELECT AVG(PRICE)

FROM PRODUCT;



SELECT B.GROUP\_NAME, AVG(PRICE)

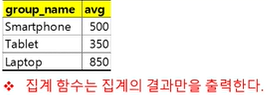
FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON (A.GROUP\_ID = B.GROUP\_ID)

GROUP BY

B.GROUP\_NAME;



SELECT A.PRODUCT\_NAME, A.PRICE, B.GROUP\_NAME,

AVG(A.PRICE) OVER(PARTITION BY B.GROUP\_NAME) //GROUP\_NAME 컬럼 기준 PRICE 평균값 출력

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON (A.GROUP\_ID = B.GROUP\_ID);



SELECT A.PRODUCT\_NAME, A.PRICE, B.GROUP\_NAME,

AVG(A.PRICE) OVER(PARTITION BY B.GROUP\_NAME ORDER BY B.GROUP\_NAME)

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON (A.GROUP\_ID = B.GROUP\_ID);

결과값 위와 동일

SELECT A.PRODUCT\_NAME, A.PRICE, B.GROUP\_NAME,

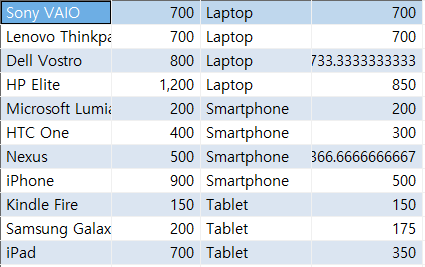
AVG(A.PRICE) OVER(PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE)

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON (A.GROUP\_ID = B.GROUP\_ID);

누적집계 출력



한개일 때 700, 두 개일때 1400/2 = 700, 세개일때 2200/3 = 733.3333333, 네 개일때 3200/4=850

**ROW\_NUMBER, RANK, DENSE\_RANK 함수 : 특정 집합 내에서 결과 건수의 변화 없이 해당 집합 안에서 특정 컬럼의 순위를 구하는 함수이다.**

**<ROW\_NUMBER 함수>**

SELECT A.PRODUCT\_NAME, B.GROUP\_NAME, A.PRICE,

ROW\_NUMBER () OVER (PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE) --순위를 구할 때 GROUP\_NAME 컬럼 기준으로 구하고 GROUP\_NAME 기준의 각 순위는 PRICE 컬럼 기준으로 정렬한다.

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON A.GROUP\_ID = B.GROUP\_ID



같은 순위가 있어도 무조건 순차적으로 순위를 매긴다.

**<RANK 함수> : 같은 순위면 같은 순위면서 다음 순위 건너뜀**

SELECT A.PRODUCT\_NAME, B.GROUP\_NAME, A.PRICE,

RANK () OVER (PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE)

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON A.GROUP\_ID = B.GROUP\_ID



**<DENSE\_RANK 함수> : 같은 순위면 같은 순위면서 다음 순위 건너뛰지 않음**

SELECT A.PRODUCT\_NAME, B.GROUP\_NAME, A.PRICE,

DENSE\_RANK () OVER (PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE)

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON A.GROUP\_ID = B.GROUP\_ID

****

**<FIRST\_VALUE 함수>**

SELECT A.PRODUCT\_NAME, B.GROUP\_NAME, A.PRICE,

**FIRST\_VALUE** (A.PRICE) OVER (PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE)

AS LOWEST\_PRICE\_PER\_GROUP

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON A.GROUP\_ID = B.GROUP\_ID

****

**<LAST\_VALUE 함수>**

SELECT A.PRODUCT\_NAME, B.GROUP\_NAME, A.PRICE,

**LAST\_VALUE** (A.PRICE) OVER (PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE

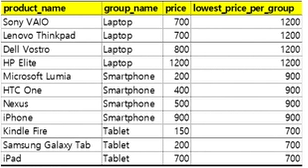
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDING FOLLOWING) AS LOWEST\_PRICE PER GROUP –GROUP\_NAME 컬럼 기준으로 PRICE 컬럼으로 정렬한 값 중에서 파티션의 첫번째 로우부터 파티션의 마지막 로우까지 가장 마지막에 나오는 PRICE 값을 출력한다.

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP\_B

ON A.GROUP\_ID = B.GROUP\_ID

LAST\_VALUE 함수 OPTION DEFAULT 값 : “RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW” 따라서 OPTION을 따로 지정해주지않으면 BOUNDARY의 마지막 값이 아닌 현재 로우의 각각의 값이 출력된다.



**LAG, LEAD 함수 : 특정 집합 내에서 결과 건수의 변화 없이 해당 집합 안에서 특정 컬럼의 이전 행의 값 혹은 다음 행의 값을 구한다.**

**<LAG 함수> : 이전행의 값을 찾는다.**

SELECT A.PRODUCT\_NAME, B.GROUP\_NAME, A.PRICE

,**LAG**(A.PRICE, 1) OVER (PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE) AS PREV\_PRICE

, A.PRICE – **LAG**(PRICE, 1) OVER (PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE) AS CUR\_PREV\_DIFF

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON A.GROUP\_ID = B.GROUP\_ID



**<LEAD 함수> : 다음행의 값을 찾는다.**

SELECT A.PRODUCT\_NAME, B.GROUP\_NAME, A.PRICE

,**LEAD**(A.PRICE, 1) OVER (PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE) AS PREV\_PRICE

, A.PRICE – **LEAD**(PRICE, 1) OVER (PARTITION BY B.GROUP\_NAME ORDER BY A.PRICE) AS CUR\_PREV\_DIFF

FROM PRODUCT A

INNER JOIN PRODUCT\_GROUP B

ON A.GROUP\_ID = B.GROUP\_ID

****