3주차

4.8 GROUP BY

1. WITH ROLLUP

그루핑된 그룹별로 소계를 가져오는 ROLLUP

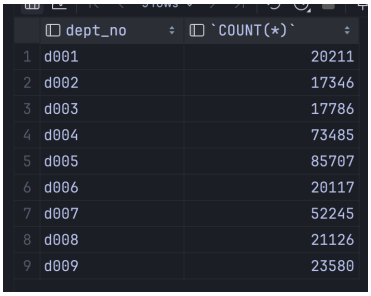
소계나 총계 체코드는 항상 해당 그룹의 마지막에 나타남

> 8.0 부터는 그룹 레코드에 표시되는 `NULL`을 사용자가 변경할 수 있게 `GROUPING()` 함수를 지원

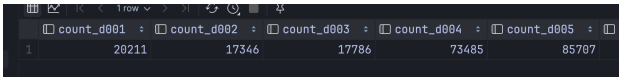
```sql
SELECT IF(GROUPING(first_name), 'All first_name', first_name) AS first_name,
       IF(GROUPING(last_name), 'All last_name', last_name)    AS last_name,
       COUNT(*)
FROM employees
GROUP BY first_name, last_name
WITH ROLLUP;
```

2. 레코드를 컬럼으로 변환하여 조회

레코드를 컬럼으로 변환



	dept_no	`COUNT(*)`
1	d001	20211
2	d002	17346
3	d003	17786
4	d004	73485
5	d005	85707
6	d006	20117
7	d007	52245
8	d008	21126
9	d009	23580



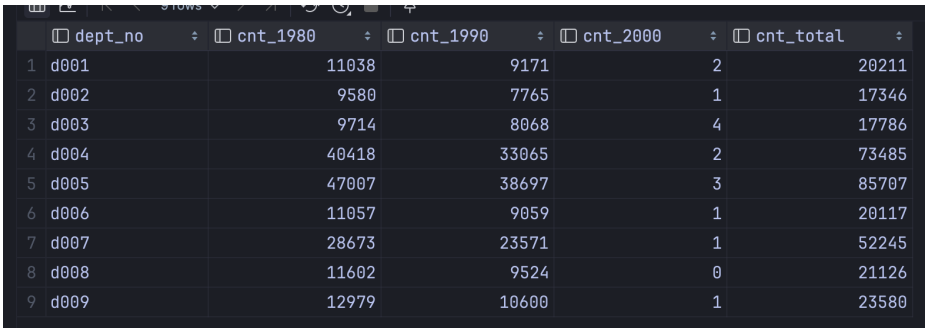
	count_d001	count_d002	count_d003	count_d004	count_d005
1	20211	17346	17786	73485	85707

```sql
SELECT SUM(CASE WHEN dept_no = 'd001' THEN emp_count ELSE 0 END) AS count_d001,
       SUM(CASE WHEN dept_no = 'd002' THEN emp_count ELSE 0 END) AS count_d002,
       SUM(CASE WHEN dept_no = 'd003' THEN emp_count ELSE 0 END) AS count_d003,
       SUM(CASE WHEN dept_no = 'd004' THEN emp_count ELSE 0 END) AS count_d004,
       SUM(CASE WHEN dept_no = 'd005' THEN emp_count ELSE 0 END) AS count_d005,
       SUM(CASE WHEN dept_no = 'd006' THEN emp_count ELSE 0 END) AS count_d006,
       SUM(CASE WHEN dept_no = 'd007' THEN emp_count ELSE 0 END) AS count_d007,
       SUM(CASE WHEN dept_no = 'd008' THEN emp_count ELSE 0 END) AS count_d008,
       SUM(CASE WHEN dept_no = 'd009' THEN emp_count ELSE 0 END) AS count_d009,
```

```
        SUM(emp_count)                                                  AS count_total
FROM (SELECT dept_no, COUNT(*) AS emp_count
      FROM dept_emp
      GROUP BY dept_no) tb_derived;
```

`dept_no`가 쿼리 일부로 사용되기 때문에 `dept_no`가 변경되는 경우 쿼리까지 변경해야 하는 문제

하나의 컬럼을 여러 컬럼으로 분리



	dept_no	cnt_1980	cnt_1990	cnt_2000	cnt_total
1	d001	11038	9171	2	20211
2	d002	9580	7765	1	17346
3	d003	9714	8068	4	17786
4	d004	40418	33065	2	73485
5	d005	47007	38697	3	85707
6	d006	11057	9059	1	20117
7	d007	28673	23571	1	52245
8	d008	11602	9524	0	21126
9	d009	12979	10600	1	23580

```
SELECT de.dept_no,
       SUM(CASE WHEN e.hire_date BETWEEN '1980-01-01' AND '1989-12-31' THEN 1 EL
       SUM(CASE WHEN e.hire_date BETWEEN '1990-01-01' AND '1999-12-31' THEN 1 EL
       SUM(CASE WHEN e.hire_date BETWEEN '2000-01-01' AND '2009-12-31' THEN 1 EL
       COUNT(*)
FROM dept_emp de,
     employees e
WHERE e.emp_no = de.emp_no
GROUP BY de.dept_no;
```

간단한 SQL 문장으로 상당히 많은 프로그램 코드를 줄이는 것이 가능하다.

4.9 ORDER BY

`ORDER BY` 절이 사용되지 않은 `SELECT` 쿼리

- 인덱스를 사용한 `SELECT`의 경우, 인덱스에 정렬된 순서대로 레코드 조회
- 풀 테이블 스캔을 실행하는 `SELECT`의 경우
  - MyISAM은 테이블에 저장된 순서대로 조회하는데 이 순서가 정확히 INSERT된 순서는 아닐 수 있다. 테이블의 레코드가 삭제되면서 빈 공간이 생기고 해당 빈 공간에 먼저 저장하기 때문
  - InnoDB는 항상 PK로 클러스터링 되어 있기 때문에 풀 테이블 스캔의 경우 기본적으로 PK 순서대로 레코드를 조회
- `SELECT` 쿼리가 임시 테이블을 거쳐 처리되면 레코드의 순서를 예측하기 어렵다.

`ORDER BY` 절이 명시되지 않은 쿼리에 대해서는 어떠한 정렬도 보장되지 않기 때문에 항상 정렬이 필요한 곳에는 `ORDER BY`를 사용하자.

인덱스를 사용하지 못하는 `ORDER BY`는 추가 정렬 작업이 수행되어 실행 계획 Extra 컬럼에 Usin filesort가 표시 (filesort는 MySQL 서버가 명시적으로 정렬 알고리즘을 수행했다는 의미 정도로 이해)

`SHOW STATUS LIKE 'Sort_%'` 쿼리를 사용해 정렬 시 메모리와 디스크 파일 이용 여부를 확인 가능 ⇒ `Sort_merge_passes` 상태 값은 메모리의 버퍼(`sort_buffer_size` 시스템 변수로 설정되는 메모리 공간)와 디스크에 저장된 레코드를 몇 번 병합했는지에 대한 값

1. ORDER BY 사용법 및 주의사항

`ORDER BY 2`는 `SELECT` 되는 컬럼 중 2번째 컬럼으로 정렬을 의미

숫자가 아닌 문자열 상수를 사용하는 경우 옵티마이저가 `ORDER BY` 절 자체를 무시

2. 여러 방향으로 동시 정렬

8.0 부터 오름차순과 내림차순을 혼용하여 인덱스 생성 가능

오름차순과 내림차순 인덱스 중 하나만 있어도 적절히 인덱스를 사용해 정렬이 가능

3. 함수나 표현식을 이용한 정렬

`COS()` 함수 참고

4.10 서브쿼리

1. SELECT 절에 사용된 서브쿼리

`SELECT` 절에 사용된 서브쿼리는 내부적으로 임시 테이블을 만들거나 쿼리를 비효율적으로 실행하게 만들지는 않기 때문에 서브쿼리가 적절히 인덱스를 사용할 수 있다면 크게 주의할 사항은 없다.

일반적으로 `SELECT` 절에 서브쿼리를 사용하면 그 서브쿼리는 항상 컬럼과 레코드가 하나인 결과를 반환해야 한다.

- 서브쿼리의 결과가 0건이면 결과는 NULL로 채워져서 반환
- 서브쿼리의 결과가 2건 이상의 레코드를 반환하는 경우 에러가 발생하며 쿼리가 종료
- 서브쿼리가 2개 이상의 컬럼을 반환하는 경우도 에러 발생

오직 스칼라 서브쿼리만 사용 가능!

> **서브쿼리로 실행될 때보다 조인으로 처리할 때가 조금 더 빠르기 때문에 가능하면 조인으로 쿼리를 작성하자**

8.0 부터 도입된 래터럴 조인을 활용하면 동일한 레코드의 각 컬럼을 가져오기 위해 서브쿼리를 여러번 남용하지 않아도 된다.

> 래터럴 조인을 사용한 경우 인덱스를 이용해 충분히 정렬된 결과를 가져올 수 있음에도 불구하고 정렬 알고리즘을 실행하는 오류가 있음. 아직 해결 안 된 듯
>
> https://bugs.mysql.com/bug.php?id=94903

2. FROM 절에 사용된 서브쿼리

기존에는 FROM 절에 서브쿼리가 사용되면 항상 서브쿼리의 결과를 임시 테이블로 저장했지만 5.7 부터는 옵티마이저가 FROM 절의 서브쿼리를 외부 쿼리로 병합하는 최적화를 수행하도록 개선

FROM 절의 서브쿼리가 외부 쿼리로 병합되지 못하는 조건

- 집합 함수 사용(SUM(), MIN(), MAX(), COUNT() 등)
- DISTINCT
- GROUP BY • HAVING
- LIMIT
- UNION(UNION DISTINCT) • UNION ALL
- SELECT 절에 서브쿼리가 사용된 경우
- 사용자 변수 사용(사용자 변수에 값이 할당되는 경우)

3. WHERE 절에 사용된 서브쿼리

동등 또는 크다 작다 비교

`= (subquery)`

5.5 부터는 서브쿼리를 먼저 실행해 상수로 변환하고 나머지 쿼리 부분을 처리한다.

튜플 형태 비교는 외부 쿼리가 인덱스를 사용하지 못하고 풀 테이블 스캔을 실행하니 주의

IN 비교

`IN (subquery)`

세미 조인

5.6 부터 세미 조인의 최적화가 많이 개선되었다. 쿼리 특성이나 조인 관계에 맞게 5개의 최적화 전략을 선택적으로 사용

- Table Pull-out
- Firstmatch
- Loosescan
- Materialization(구체화)
- Duplicated Weed-out(중복 제거)

NOT IN 비교

`NOT IN (subquery)`

안티 세미 조인

최적화 전략

- NOT EXISTS
- Materialization(구체화)

두 방법 모두 성능 향상에 크게 도움이 되지 않음. WHERE 절에 단독으로 안티 세미 조인 조건만 존재하면 풀 테이블 스캔을 피할 수 없음

4.11 Common Table Expression(CTE)

이름을 가지는 임시 테이블. SQL 문장 내에서 한 번 이상 사용될 수 있으며 SQL 문장이 종료되면 자동으로 삭제됨

재귀적 반복 실행 여부 기준 2가지로 분류

- Non-recursive
- Recursive

사용 가능 위치

- SELECT, UPDATE, DELETE 문장 제일 앞
- 서브쿼리의 제일 앞
- SELECT 절의 바로 앞

1. Non-Recursive CTE

WITH 절을 이용해 CTE 정의

```
WITH cte1 AS (SELECT * FROM departments)
SELECT *
FROM cte1;
```

cte1이 한 번만 사용되기 때문에 FROM 절의 서브쿼리로 바꿔 사용 가능 (실제 두 쿼리는 실행 계획까지 동일)

```
SELECT *
FROM (SELECT * FROM departments) cte1;
```

여러 개의 CTE 임시 테이블을 사용하는 쿼리도 FROM 절의 서브쿼리로 대체해서 사용할 수 있지만 실행 계획이 달라진다. CTE를 이용한 쿼리에서는 임시 테이블을 한 번만 생성하지만, 서브쿼리를 이용하면 n개의 임시 테이블을 생성한다.

CTE 쿼리에서 이전에 CTE로 생성된 임시 테이블을 참조 가능

서브쿼리와 비교하였을 때 장점

- CTE 임시 테이블은 재사용 가능하므로 FROM 절의 서브쿼리보다 효율적
- CTE로 선언된 임시 테이블을 다른 CTE 쿼리에서 참조 가능
- CTE는 임시 테이블의 생성 부분과 사용 부분의 코드를 분리할 수 있어 높은 가독성 제공

2. Recursive CTE

```
WITH RECURSIVE cte (no) AS (SELECT 1
                            UNION ALL
                            SELECT (no + 1)
                            FROM cte
                            WHERE no < 5)
```

```
SELECT *
FROM cte;
```

재귀적 CTE 쿼리는 비 재귀적 쿼리 파트와 재귀적 파트로 구분하며 반드시 이 둘을 `UNION`(`UNION DISTINCT`) 또는 `UNION ALL`로 연결하는 형태로 쿼리를 작성해야 한다.

`SELECT 1`이 비 재귀적 파트. `SELECT (no + 1) FROM cte WHERE no < 5`이 재귀적 파트.

비 재귀적 파트는 처음 한 번만 실행, 재귀적 파트는 쿼리 결과가 없을 때까지 반복 실행

쿼리 작동 순서

1. CTE 쿼리의 비 재귀적 파트의 쿼리 실행 ⇒ `SELECT 1`
2. 1번 결과를 이용해 cte라는 이름의 임시 테이블 생성
3. 1번 결과를 cte라는 임시 테이블에 저장
4. 1번 결과를 입력으로 사용해 CTE 쿼리의 재귀적 파트의 쿼리 실행 ⇒ `SELECT (no + 1) FROM cte WHERE no < 5`
5. 4번 결과를 cte라는 임시 테이블에 저장(이때 UNION 또는 UNION DISTINCT의 경우 중복 제거 실행)
6. 전 단계의 결과를 입력으로 사용해 CTE 쿼리의 재귀적 파트 쿼리 실행
7. 6번 단계 쿼리 결과가 없으면 CTE 종료
8. 6번 결과를 cte라는 임시 테이블에 저장
9. 6번으로 돌아가 반복 실행

> **CTE 임시 테이블의 구조(컬럼명, 데이터 타입)는 CTE 쿼리의 비 재귀적 쿼리 파트의 결과로 결정**

재귀적 쿼리 파트를 실행할 때는 지금까지의 모든 단계에서 만들어진 결과 set이 아닌 직전 단계의 결과만 재귀 쿼리의 입력으로 사용 (1, 2, 3이 생성된 이후 실행되는 재귀 쿼리의 입력은 3)

재귀 쿼리가 반복을 종료하는 조건은 재귀 파트 쿼리의 결과가 0건일 때

무한 반복 경우를 막기 위해 `cte_max_recursion_depth` 시스템 변수를 이용해 최대 반복 실행 횟수 제한 가능(기본값 1000. 적절히 낮은 값으로 변경하고, 꼭 필요한 쿼리에서만 `SET_VAR` 힌트를 사용해 해당 쿼리에서만 반복 호출 횟수를 늘리는 방법 권장)

3. Recursive CTE 활용

```
WITH RECURSIVE managers AS (SELECT *, CAST(id AS CHAR(100)) AS manager_path, 1 A
                            FROM employees
                            WHERE manager_id IS NULL
                            UNION ALL
                            SELECT e.*, CONCAT(e.id, ' -> ', m.manager_path) AS
                            FROM managers m
                                     INNER JOIN employees e ON e.manager_id = m.
SELECT *
```

```
FROM managers
ORDER BY lv;
```

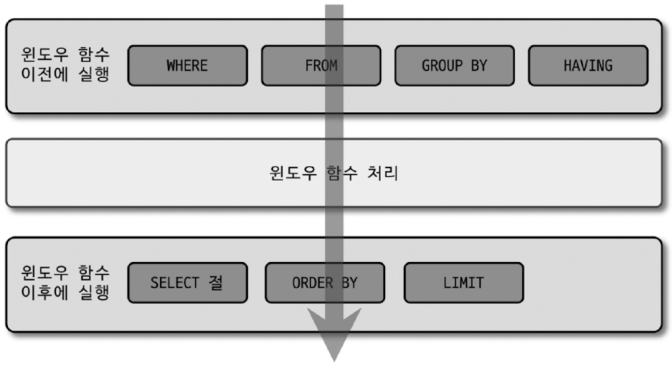
> 재귀적 쿼리는 혼란스러울 수 있지만 작동 원리만 이해하면 어렵지 않게 자유자재로 쿼리를 작성할 수 있을 것이다! 공부해 두면 도움이 많이 될 것이다.

4.12 윈도우 함수(Window Function)

윈도우 함수: 조회하는 현재 레코드를 기준으로 연관된 레코드 집합의 연산을 수행

> **아직 이해가 잘 안 되는 부분**
> 일반적인 SQL 문장에서 하나의 레코드를 연산할 때 다른 레코드의 값을 참조할 수 없는데, 예외적으로 GROUP BY 또는 집계 함수를 이용하면 다른 레코드의 컬럼값을 참조할 수 있다. 하지만 GROUP BY 또는 집계 함수를 사용하면 결과 집합의 모양이 바뀐다. 그에 반해, 윈도우 함수는 결과 집합을 그대로 유지하면서 하나의 레코드 연산에 다른 레코드의 컬럼값을 참조할 수 있다.

1. 쿼리 각 절의 실행 순서



쿼리 각 절의 실행 순서

윈도우 함수를 사용하는 쿼리의 결과 레코드는 `FROM` 절과 `WHERE` 절, `GROUP BY`와 `HAVING` 절에 의해 결정 ⇒ 윈도우 함수를 `GROUP BY` 컬럼으로 사용하거나 `WHERE` 절에 사용할 수 없음. `LIMIT`을 적용하고 윈도우 함수를 적용할 수 없음. (`FROM` 절에 서브쿼리를 사용하면 됨)

2. 윈도우 함수 기본 사용법

```
AGGREGATE_FUNC() OVER(<partition> <order>) AS window_func_column
```

직원들의 입사 순서를 조회하는 쿼리

```
SELECT *, RANK() OVER (ORDER BY e.hire_date) AS hire_date_rank
```

```
FROM employees e;
```

부서별 직원 입사 순서를 조회하는 쿼리

```
SELECT de.dept_no,
       e.emp_no,
       e.first_name,
       e.hire_date,
       RANK() OVER (PARTITION BY de.dept_no ORDER BY e.hire_date) AS hire_date_r
FROM employees e
         INNER JOIN dept_emp de ON de.emp_no = e.emp_no
ORDER BY de.dept_no, e.hire_date;
```

프레임: 윈도우 함수의 각 파티션 안에서도 연산 대상 레코드별로 연산을 소행할 소그룹

윈도우 함수에서 프레임을 명시적으로 지정하지 않아도 MySQL 서버는 상황에 맞게 묵시적으로 선택. 프레임은 레코드의 순서대로 현재 레코드 기준 앞뒤 몇 건을 연산 범위로 제한하는 역할

```
AGGREGATE_FUNC() OVER(<partition> <order> <frame_clause>) AS window_func_column

frame_clause:
    frame_units frame_extent

frame_units:
    {ROWS | RANGE}

frame_extent:
    {frame_start | frame_between}

frame_between:
    BETWEEN frame_start AND frame_end

frame_start, frame_end: {
    CURRENT ROW
  | UNBOUNDED PRECEDING
  | UNBOUNDED FOLLOWING
  | expr PRECEDING
  | expr FOLLOWING
}
```

frame_units

- `ROWS`: 레코드의 위치를 기준으로 프레임 생성
- `RANGE`: ORDER BY 절에 명시된 컬럼을 기준으로 값의 범위로 프레임 생성

frame_start, frame_end

- `CURRENT ROW`: 현재 레코드
- `UNBOUNDED PRECEDING`: 파티션의 첫 번째 레코드
- `UNBOUNDED FOLLOWING`: 파티션의 마지막 레코드
- `expr PRECEDING`: 현재 레코드로부터 n번째 이전 레코
- `expr FOLLOWING`: 현재 레코드로부터 n번째 이후 레코드

> 윈도우 함수에서 프레임이 별도로 명시되지 않는 경우
>
> `OVER()` 절이 `ORDER BY`를 가지는 경우, 파티션의 첫 번째 레코드부터 현재 레코드까지
>
> `ORDER BY`를 가지지 않는 경우 모든 레코드

자동 프레임이 파티션의 전체 레코드로 고정된 함수들

- `CUME_DIST()`
- `DENSE_RANK()`
- `LAG()`
- `LEAD()`
- `NTILE()`
- `PRECENT_RANK()`
- `RANK()`
- `ROW_NUMBER()`

3. 윈도우 함수

aggregate functions

GROUP BY 절과 사용되며, OVER() 없이 단독으로 사용 가능

MySQL :: MySQL 8.0 Reference Manual :: 14.19.1 Aggregate Function Descriptions
This section describes aggregate functions that operate on sets of values. They are often used with a GROUP BY clause to group values into subsets.
https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html

non-aggreagte functions

반드시 OVER() 절을 가지고 있어야 하며 윈도우 함수로만 사용 가능

MySQL :: MySQL 8.0 Reference Manual :: 14.20.1 Window Function Descriptions

This section describes nonaggregate window functions that, for each row from a query, perform a calculation using rows related to that row. Most aggregate functions also can be used as window

https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html

4. 윈도우 함수와 성능

8.0 부터 도입되었으며 아직 인덱스를 이용한 최적화가 부족한 부분도 있다.

가능하다면 윈도우 함수에 너무 의존하지 않는 것이 좋다.

4.13 잠금을 사용하는 SELECT

FOR SHARE

`SELECT` 쿼리로 읽은 레코드에 대해서 읽기 잠금(공유 잠금, Shared lock)
다른 트랜잭션에서 해당 레코드를 변경하지 못한다. 읽기는 가능

FOR UPDATE

`SELECT` 쿼리가 읽은 레코드에 대해 쓰기 잠금(배타 잠금, Exclusive lock)
다른 트랜잭션에서 레코드를 변경하는 것 뿐 아니라 읽기도 수행 불가능

`SELECT … FOR UPDATE` 쿼리에 의해 잠겨진 상태라 하더라도 단순 `SELECT` 쿼리는 아무런 대기 없이 실행

1. 잠금 테이블 선택

`FOR UPDATE` 뒤에 `OF table_name` 절을 추가하여 해당 테이블에 대해서만 잠금을 걸도록 설정 가능

```
SELECT *
FROM employees e
         INNER JOIN dept_emp de ON de.emp_no = e.emp_no
         INNER JOIN departments d ON d.dept_no = de.dept_no
WHERE e.emp_no = 10001 FOR
UPDATE OF e;
```

2. NOWAIT & SKIP LOCKED

`FOR SHARE` 또는 `FOR UPDATE` 뒤에 `NOWAIT` 옵션을 추가하면, 해당 레코드가 다른 트랜잭션에 의해 잠겨진 상태라면 에러를 반환하며 쿼리가 즉시 종료된다.

`SKIP LOCKED` 옵션은 `SELECT` 하려는 레코드가 다른 트랜잭션에 의해 이미 잠겨진 상태라면 잠긴 레코드는 무시하고 잠금이 걸리지 않은 레코드만 가져온다.

> `SKIP LOCKED` 절을 가진 `SELECT` 구문은 NOT-DETERMINISTIC 쿼리가 된다. 이는 쿼리를 실행하는 시점에 따라 각 트랜잭션의 간섭에 의해 다른 결과를 반환할 수 있다. 때문에 문장(STATEMENT) 기반의 복제에서 소스 서버와 레플리카 서버의 데이터를 다르게 만들 수 있으므로 가능하면 복제의 바이너리 로그 포맷으로 STATEMENT보다는 ROW 또는 MIXED를 사용하자.