AWS Aurora 환경에서 PostgreSQL 의 PG_HINT_PLAN 사용 방법

Author	이경오
Creation Date	
Last Updated	
Version	

Version	변경일자	변경자(작성자)	주요내용
1			
2			
3			
4			

Contents

1. 본 노트의 목적

세계적으로 널리 사용되고 있는 RDBMS 인 Oracle 은 강력한 SQL Hint 기능을 가지고 있습니다. SQL 힌트에 대해 충분한 학습이 된 개발자 및 SQL Tuner 는 Hint 를 사용해서 실행계획을 자유자재로 변경 및 조절을 할수 있습니다. 이러한 Oracle 의 강력한 기능에 익숙한 사용자는 PostgreSQL 을 만나면 매우 당황을 하게 되는데 그 이유는 PostgreSQL 은 공식적으로 SQL Hint 기능을 지원하지 않기 때문입니다. 하지만 PG_HINT_PLAN 이라는 Extension 이 존재하며 해당 Extension 을 사용하면 PostgreSQL 에서도 SQL Hint 사용을 통한 실행계획 제어가 가능합니다. 물론 Oracle 만큼의 다양한 SQL Hint 가 존재하는 것은 아니지만 기본적인 실행계획 제어를 Hint 는 모두 존재합니다. 즉 대부분의 성능 저하 SQL 문의 튜닝이 가능한 것입니다.

Oracle 과 마찬가지로 PostgreSQL 에서도 SQL Hint 는 SQL 실행계획에 영향을 주는 명령어라고 할 수 있습니다. PostgreSQL 에서는 SQL Hint 가 Plan Tree 자체를 바꾸므로 절대로 무시되지 않습니다. 즉 매우 강력한 실행계획 기능이면서도 잘못 적용할 시에는 오히려 성능에 악영향을 미칠 수 있기 때문에 적용 전에 신중을 기해야 합니다. 하지만 SQL 실행계획 분석에 의한 성능 개선에 익숙한 사용자 혹은 SQL Tuner 라면 매우 강력하고 유용한 기능이 될 수 있습니다.

이번 노트에서는 PG_HINT_PLAN 을 이용한 실행계획 제어에 대해서 알아볼 것입니다. 또한 최근 각광받고 있는 AWS Aurora 환경의 PostgreSQL 에서 실습을 진행할 것입니다.

또한 이 노트에서는 SQL Tuning 지식의 기본적인 사항을 이미 숙지하고 있는 것으로 간주합니다. (예: Index Scan 과 Table Full Scan 의 차이점등에 대해서는 별도로 설명하지 않습니다.)

2. PG_HINT_PLAN사용을 위한 설치

AWS Autora 의 PostgreSQL 환경에서 PG_HINT_PLAN 을 사용하기 위한 설치를 진행합니다. 실습을 진행하는 PostgresSQL 의 버전은 아래와 같습니다.

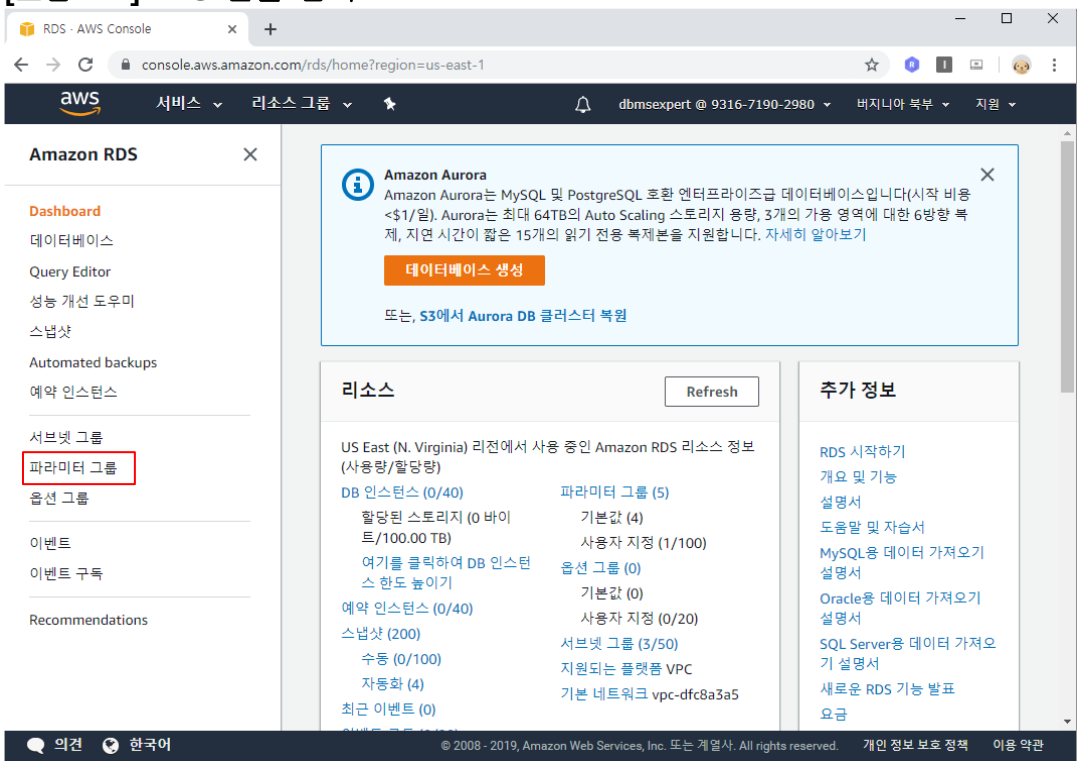
```
PostgreSQL 10.7 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.9.3, 64-bit
```

그럼 지금부터 PG_HINT_PLAN Extension 을 설치하도록 합니다.

2.1. AWS RDS 파라미터 그룹 변경

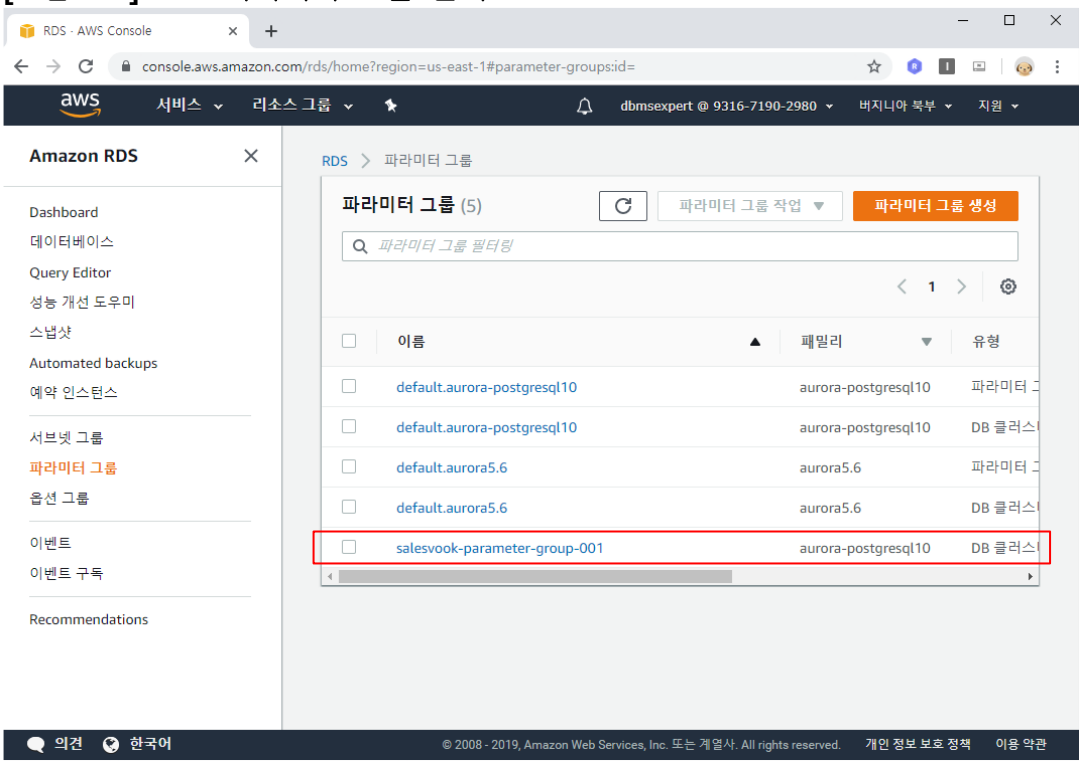
우선 사용중인 AWS RDS 콘솔에 접속합니다.

[그림 2-1] RDS 콘솔 접속



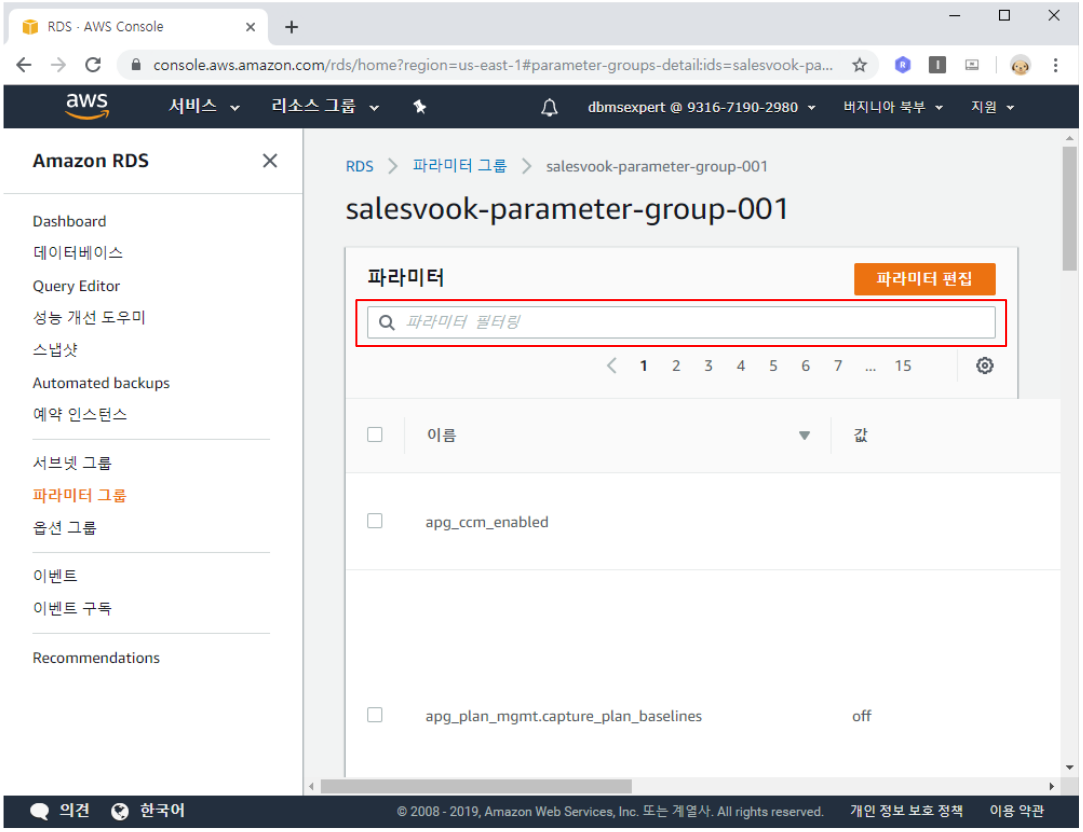
접속에 성공하면 왼쪽 메뉴의 "파라미터 그룹"을 클릭합니다. "파라미터 그룹"은 해당 AWS RDS 내에 존재하는 PostgreSQL DBMS 의 파라미터 설정을 할 수 있는 메뉴입니다.

[그림 2-2] RDS 파라미터 그룹 선택



현재 사용중인 파라미터 그룹을 선택합니다. 본 노트의 작성자는 "salesvook-parameter-group-001"을 선택하였습니다. 해당 RDS 가 현재 사용하고 있는 파라미터 그룹입니다.

[그림 2-3] RDS 파라미터 변경



지금부터 파라미터에 대한 변경이 가능합니다. 아래의 표와 같이 파라미터를 변경합니다. PostgreSQL 의 각종 파라미터를 변경할 수 있습니다. 해당 파라미터 그룹 내에 존재하는 파라미터를 변경하면 해당 파라미터 그룹은 RDS 내에 PostgreSQL 의 설정을 바꾸게 됩니다.

[표 2-1] PG_HINT_PLAN 사용을 위한 파라미터 설정

파라미터명	값
shared_preload_libraries	pg_hint_plan 값을 추가
pg_hint_plan.enable_hint	1 로 설정
pg_hint_plan.debug_print	off 로 설정
pg_hint_plan.message_level	info 로 설정

해당 파라미터를 변경 후 "Amazon RDS" -> "데이터베이스" -> "수정"을 수행하여 파라미터 그룹을 적용시킵니다. 즉 해당 데이터베이스를 재시작하면 적용됩니다. 운영중에 해당 작업을 수행할 때는 현재 업무가 처리중이라면 매우 주의해서 작업해야 합니다.

2.2. Extension 추가

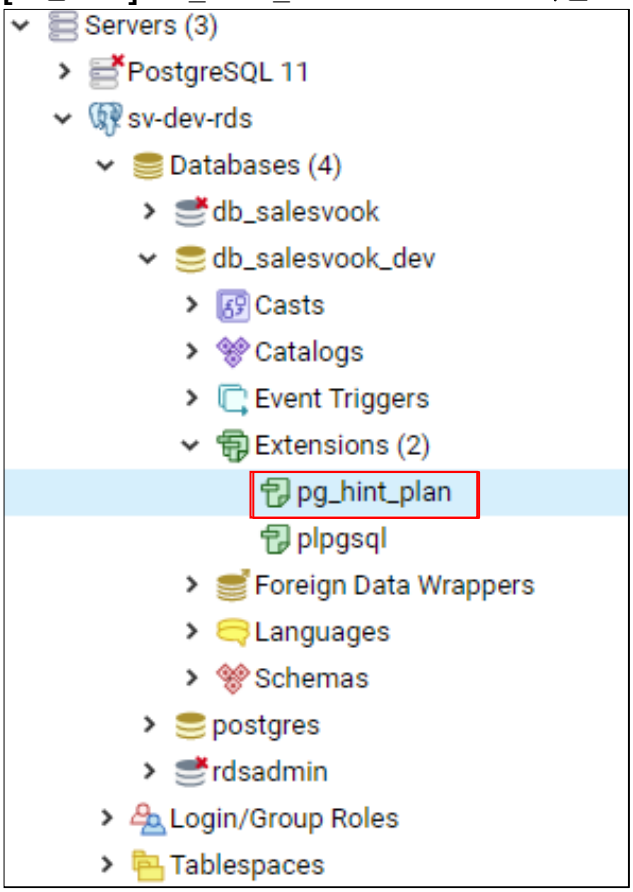
파라미터 추가/수정 및 적용이 완료되면 아래의 명령으로 PG_HINT_PLAN Extension 을 추가합니다.

[스크립트 2-1] PG_HINT_PLAN 사용을 위한 Extension 추가

```
--익스텐션 추가
create extension pg_hint_plan;
```

익스텐션 추가 후 PG_ADMIN 에서 해당 Extension 이 정상적으로 추가되었는지 확인합니다. PG_HINT_PLAN Extension 의 추가가 가능한 이유는 위에서 shared_preload_libraries 파라미터에 PG_HINT_PLAN 관련 값을 Setting 시켰기 때문에 가능한 것입니다.

[그림 2-4] PG_HINT_PLAN Extension 확인



정상적으로 추가된 것을 확인하였습니다. 지금부터 PG_HINT_PLAN 을 이용하여 SQL Hint 를 사용할 수 있습니다.

3. PG_HINT_PLAN의 활용

3.1. 기본 사용법

SQL Hint 의 기본 사용법은 아래와 같습니다. 우선 아래의 SQL 문을 보도록 합니다.

[스크립트 3-1] SQL Hint 사용 전

```
select
         a.customer_id
       , b.rental_id
       , b.return_date
       , c.payment_date
       , c.amount
 from
         public.customer a
       , public.rental b
       , public.payment c
where 1=1
  and a.customer_id = 388
  and a.customer_id = b.customer_id
  and b.rental_id = c.rental_id;
```

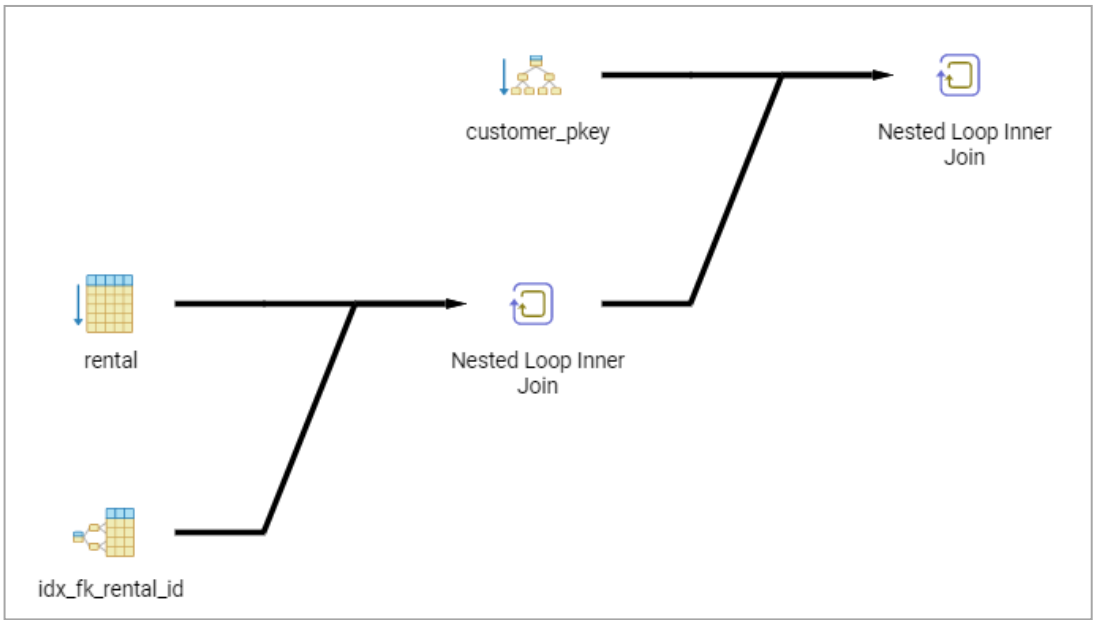
customer 테이블에서 고객 아이디가 '388'인 고객의 렌탈 및 지불 정보를 조회하는 SQL 문입니다. 해당 SQL 문은 일반적인 업무에게 매우 자주 쓰이는 SQL 문의 유형입니다. (1:M 관계)

이 SQL 문의 실행계획을 살펴보면 아래와 같습니다.

```
Nested Loop  (cost=0.56..543.21 rows=24 width=30)
  ->  Index Only Scan using customer_pkey on customer a
(cost=0.28..8.29 rows=1 width=4)
       Index Cond: (customer_id = 388)
  ->  Nested Loop  (cost=0.29..534.67 rows=24 width=28)
       ->  Seq Scan on rental b  (cost=0.00..350.55 rows=26 width=14)
            Filter: (customer_id = 388)
       ->  Index Scan using idx_fk_rental_id on payment c
(cost=0.29..7.07 rows=1 width=18)
            Index Cond: (rental_id = b.rental_id)
```

실행계획을 알아보기 쉽게 순서도로 표현하면 아래와 같습니다.

[그림 3-1] 실행 계획 순서도



우선 rental 테이블을 풀 스캔하고 payment 테이블의 idx_fk_rental_id 인덱스와 Nested Loop 조인을 수행하면서 customer 테이블의 customer_pkey 인덱스와 Nested Loop 조인을 수행하고 있습니다.

그럼 지금부터 아래와 같이 SQL Hint 중 하나인 Leading 힌트를 이용하여 customer 테이블 혹은 Rentral 테이블을 드라이빙 테이블로 하고 payment 과 조인될 수 있도록 조인 순서를 조정해보도록 하겠습니다.

[스크립트 3-2] SQL Hint 사용 후

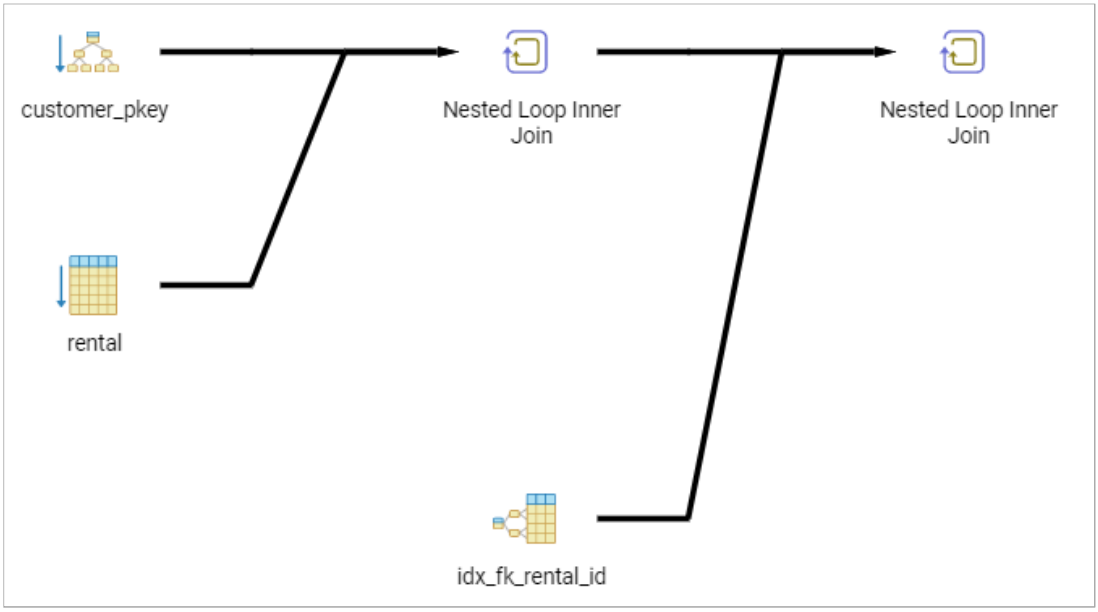
```
/*+
       Leading(a b c)
*/
select
         a.customer_id
       , b.rental_id
       , b.return_date
       , c.payment_date
       , c.amount
 from
         public.customer a
       , public.rental b
       , public.payment c
where 1=1
  and a.customer_id = 388
  and a.customer_id = b.customer_id
  and b.rental_id = c.rental_id;
```

실행계획은 아래와 같습니다.

```
Nested Loop  (cost=0.56..543.23 rows=24 width=30)
  ->  Nested Loop  (cost=0.28..359.10 rows=26 width=16)
        ->  Index Only Scan using customer_pkey on customer a
(cost=0.28..8.29 rows=1 width=4)
              Index Cond: (customer_id = 388)
        ->  Seq Scan on rental b  (cost=0.00..350.55 rows=26 width=14)
              Filter: (customer_id = 388)
  ->  Index Scan using idx_fk_rental_id on payment c  (cost=0.29..7.07
rows=1 width=18)
        Index Cond: (rental_id = b.rental_id)
```

customer 테이블(customer_pkey 인덱스만 스캔함)을 먼저 읽은 후 rental 테이블과 Nested Loop 조인하고 다시 payment 테이블과 Nested Loop 조인한 것을 알 수 있습니다. 이 실행계획을 순서도로 표현하면 아래와 같습니다.

[그림 3-2] 실행 계획 순서도



즉 SQL Hint 를 사용함으로써 SQL 문의 실행계획을 바꾸는 것을 알수 있습니다. 즉 PostgreSQL 도 Oracle 과 유사하게 SQL Hint 를 통한 실행계획 조정이 가능한 것을 알 수 있습니다.

3.2. Hint의 종류

PG_HINT_PLAN 이 제공하는 주요 SQL Hint 는 아래 표와 같습니다.

[표 3-1] PG_HINT_PLAN 주요 힌트

힌트명	설명
SeqScan (table)	- Seq Scan 방식으로 유도함
IndexScan (table index)	- Index Scan 방식으로 유도함

IndexOnlyScan (table index)	- Index Only Scan 방식으로 유도함 - 만일 Index Only Scan 을 사용할 수 없다면, Index Scan 방식을 사용함
BitmapScan (table index)	- Bitmap Scan 방식으로 유도함
NoIndexScan (table)	- Index Scan 과 Index Only Scan 방식을 사용하지 않도록 함
NoIndexOnlyScan (table)	- Index Only Scan 방식을 사용하지 않도록 함
NoBitmapScan (table)	- Bitmap Scan 방식을 사용하지 않도록 함
NestLoop (table table)	- Nested Loop 조인으로 유도함
HashJoin (table table)	- 해시 조인으로 유도함
MergeJoin (table table)	- Merge 조인으로 유도함
NoNestLoop (table table)	- Nested Loop 조인을 사용하지 않도록 함
NoHashJoin (table table)	- 해시 조인을 사용하지 않도록 함
NoMergeJoin (table table)	- Merge 조인을 사용하지 않도록 함

위 표에서 'table'은 테이블명 혹은 테이블 Alias 를 뜻하고 'index'는 인덱스명을 뜻합니다. 한가지 주의할 점은 테이블명, Alias, 인덱스명을 입력 시 소문자로 입력해야 SQL Hint 가 인식할 수 있다는 것입니다.

3.3. SeqScan 실습

아래의 SQL 문이 있습니다. 해당 SQL 문은 customer 테이블에서 customer_id 가 '388'인 행의 고객 정보를 출력하고 있습니다.

[스크립트 3-3] SeqScan Hint 사용 전

```
select
       a.customer_id
     , a.first_name
     , a.last_name
     , a.email
 from
       public.customer a
where 1=1
  and a.customer_id = 388
;
```

실행계획을 확인하면 아래와 같습니다.

```
Index Scan using customer_pkey on customer a  (cost=0.28..8.29 rows=1
width=49)
  Index Cond: (customer_id = 388)
```

customer_pkey 인덱스를 스캔하여 테이블에 접근한 것을 알 수 있습니다. 옵티마이저가 변별력이 좋은 customer_pkey 인덱스를 스캔하도록 실행계획을 생성하여 실행한 것을 알 수 있습니다. 이러한 실행계획을 테이블 풀 스캔으로 강제

로 유도할 수 있습니다.

아래와 같이 SeqScan 힌트를 이용하여 테이블 풀 스캔을 유도합니다.

[스크립트 3-4] SeqScan Hint 사용 후

```
/*+
       SeqScan(a)
 */
select
          a.customer_id
        , a.first_name
        , a.last_name
        , a.email
   from
         public.customer a
  where 1=1
    and a.customer_id = 388
;
```

실행계획을 확인하면 아래와 같습니다.

```
Seq Scan on customer a  (cost=0.00..16.49 rows=1 width=49)
  Filter: (customer_id = 388)
```

customer 테이블을 테이블 풀 스캔 하였으며 customer_id 조건을 필터 처리한 것을 알 수 있습니다.

3.4. IndexScan 실습

그럼 지금부터 IndexScan 힌트를 이용하여 인덱스 스캔을 유도하는 방법을 알아보겠습니다. 일단 아래의 SQL 을 보도록 합니다.

[스크립트 3-5] IndexScan Hint 사용 전

```
select
       a.customer_id
     , b.rental_id
     , b.rental_date
     , a.first_name
     , a.last_name
  from
       public.customer a
     , public.rental b
where 1=1
  and a.customer_id = 388
  and a.customer_id = b.customer_id
;
```

customer_id 가 388 인 고객의 고객 정보 및 렌탈 정보를 출력하고 있는 SQL 문입니다.

```
Nested Loop  (cost=4.76..80.56 rows=26 width=29)
  ->  Index Scan using customer_pkey on customer a  (cost=0.28..8.29
rows=1 width=17)
```

```
        Index Cond: (customer_id = 388)
  ->  Bitmap Heap Scan on rental b  (cost=4.49..72.01 rows=26 width=14)
        Recheck Cond: (customer_id = 388)
        ->  Bitmap Index Scan on idx_customer_id  (cost=0.00..4.48
rows=26 width=0)
              Index Cond: (customer_id = 388)
```

customer 테이블의 customer_pkey 인덱스를 스캔하여 고객 정보를 가져온 후 rental 테이블과 조인하는데 조인 시 idx_customer_id 인덱스를 Bitmap Index Scan 하고 있습니다.

그럼 지금부터 아래와 같이 SQL hint 를 써서 Bitmap Index Scan 을 일반적인 Index Scan 으로 바꿔보도록 하겠습니다.

[스크립트 3-6] IndexScan Hint 사용 후

```
/*+
        IndexScan(b idx_customer_id)
 */
select
        a.customer_id
      , b.rental_id
      , b.rental_date
      , a.first_name
      , a.last_name
  from
        public.customer a
      , public.rental b
where 1=1
  and a.customer_id = 388
  and a.customer_id = b.customer_id
;
```

IndexScan 힌트를 써서 일반적인 인덱스 스캔을 유도하였습니다. 실행계획을 살펴보면 아래와 같습니다.

```
Nested Loop  (cost=0.56..109.29 rows=26 width=29)
  ->  Index Scan using customer_pkey on customer a  (cost=0.28..8.29
rows=1 width=17)
        Index Cond: (customer_id = 388)
  ->  Index Scan using idx_customer_id on rental b  (cost=0.29..100.74
rows=26 width=14)
        Index Cond: (customer_id = 388)
```

후행 테이블인 rental 테이블을 스캔 시 일반적인 Index Scan 을 수행한 것을 알 수 있습니다. 즉 IndexScan SQL hint 가 정상적으로 작동한 것입니다.

3.5. NestLoop & IndexScan 실습

그럼 지금부터 Materialize 방식으로 Seq Scan 한후 NestLoop 조인을 하고 있는 SQL 문을 정상적인 인덱스 스캔을 통한 Nested Loop 조인 방식으로 고정하는 실습을 진행해 보도록 하겠습니다.

우선 아래의 SQL 을 살펴보도록 합니다.

[스크립트 3-5] NestLoop & IndexScan Hint 사용 전

```
select
         a.customer_id
       , b.rental_id
       , b.return_date
       , c.payment_id
       , c.payment_date
       , d.staff_id
       , d.first_name
       , d.last_name
 from
         public.customer a
       , public.rental b
       , public.payment c
       , public.staff d
where 1=1
  and a.customer_id = 388
  and a.customer_id = b.customer_id
  and b.rental_id = c.rental_id
  and c.staff_id = d.staff_id
;
```

customer_id 가 388 인 고객의 렌탈 정보와 지불정보, 담당직원 정보를 출력하고 있습니다.

그럼 실행계획을 살펴봅니다.

```
Nested Loop  (cost=5.05..266.29 rows=24 width=248)
  Join Filter: (c.staff_id = d.staff_id)
  ->  Nested Loop  (cost=5.05..264.67 rows=24 width=30)
        ->  Index Only Scan using customer_pkey on customer a
(cost=0.28..8.29 rows=1 width=4)
              Index Cond: (customer_id = 388)
        ->  Nested Loop  (cost=4.77..256.14 rows=24 width=28)
              ->  Bitmap Heap Scan on rental b  (cost=4.49..72.01 rows=26
width=14)
                    Recheck Cond: (customer_id = 388)
                    ->  Bitmap Index Scan on idx_customer_id
(cost=0.00..4.48 rows=26 width=0)
                          Index Cond: (customer_id = 388)
              ->  Index Scan using idx_fk_rental_id on payment c
(cost=0.29..7.07 rows=1 width=18)
                    Index Cond: (rental_id = b.rental_id)
  ->  Materialize  (cost=0.00..1.03 rows=2 width=220)
        ->  Seq Scan on staff d  (cost=0.00..1.02 rows=2 width=220)
```

customer, rental, payment 테이블을 조인할 시에는 정상적인 Index 스캔을 통한 Nested Loop 조인을 하였습니다. 하지만 staff 테이블은 Materialize 방식으로 가져오면서 Seq Scan 을 한 후 NestLoop 조인을 하였습니다. (반드시 해당 실행계획이 성능 상 나쁘다는 것은 아닙니다.

그럼 지금부터 NestLoop 힌트를 이용하여 staff 테이블까지 Nested Loop 조인을

유도해 보도록 하겠습니다. (보통의 경우입니다.)

[스크립트 3-6] NestLoop & IndexScan Hint 사용 후

```
/*+
        NestLoop(a b c d)
        IndexScan(d staff_pkey)
        IndexScan(b idx_customer_id)
 */
select
          a.customer_id
        , b.rental_id
        , b.return_date
        , c.payment_id
        , c.payment_date
        , d.staff_id
        , d.first_name
        , d.last_name
 from
          public.customer a
        , public.rental b
        , public.payment c
        , public.staff d
where 1=1
  and a.customer_id = 388
  and a.customer_id = b.customer_id
  and b.rental_id = c.rental_id
  and c.staff_id = d.staff_id
;
```

NestLoop 힌트를 이용하고 IndexScan 힌트도 추가적으로 이용하였습니다. 즉 일반적인 상황에서 최고의 효율을 발휘할 수 있도록 실행계획을 조절한 것입니다.

실행계획은 아래와 같습니다.

```
Nested Loop  (cost=0.97..296.90 rows=24 width=248)
  ->  Nested Loop  (cost=0.85..293.40 rows=24 width=30)
        ->  Index Only Scan using customer_pkey on customer a
(cost=0.28..8.29 rows=1 width=4)
              Index Cond: (customer_id = 388)
        ->  Nested Loop  (cost=0.57..284.86 rows=24 width=28)
              ->  Index Scan using idx_customer_id on rental b
(cost=0.29..100.74 rows=26 width=14)
                    Index Cond: (customer_id = 388)
              ->  Index Scan using idx_fk_rental_id on payment c
(cost=0.29..7.07 rows=1 width=18)
                    Index Cond: (rental_id = b.rental_id)
  ->  Index Scan using staff_pkey on staff d  (cost=0.13..0.15 rows=1
width=220)
        Index Cond: (staff_id = c.staff_id)
```

staff 테이블을 스캔시 staff_pkey 인덱스를 이용하여 Nested Loop 조인을 한 것을 알 수 있습니다.

3.6. HashJoin & SeqScan 실습

이번에는 HashJoin 및 SeqScan 힌트를 이용하여 SQL 실행계획을 테이블 풀 스캔에 의한 해시 조인으로 고정해 보도록 하겠습니다.

우선 아래의 SQL 문을 살펴보도록 합니다.

[스크립트 3-7] HashJoin & SeqScan Hint 사용 전

```
select
         a.customer_id
       , b.rental_id
       , b.return_date
       , c.payment_id
       , c.payment_date
       , d.staff_id
       , d.first_name
       , d.last_name
 from
         public.customer a
       , public.rental b
       , public.payment c
       , public.staff d
where 1=1
  and a.customer_id between 388 and 400
  and a.customer_id = b.customer_id
  and b.rental_id = c.rental_id
  and c.staff_id = d.staff_id
;
```

실행계획을 확인하면 아래와 같습니다.

```
Hash Join  (cost=10.42..477.70 rows=292 width=248)
  Hash Cond: (c.staff_id = d.staff_id)
  ->  Nested Loop  (cost=9.38..474.26 rows=292 width=30)
        ->  Hash Join  (cost=9.09..361.95 rows=321 width=16)
              Hash Cond: (b.customer_id = a.customer_id)
              ->  Seq Scan on rental b  (cost=0.00..310.44 rows=16044
width=14)
              ->  Hash  (cost=8.94..8.94 rows=12 width=4)
                    ->  Index Only Scan using customer_pkey on customer a
(cost=0.28..8.94 rows=12 width=4)
                          Index Cond: ((customer_id >= 388) AND
(customer_id <= 400))
        ->  Index Scan using idx_fk_rental_id on payment c
(cost=0.29..0.34 rows=1 width=18)
              Index Cond: (rental_id = b.rental_id)
  ->  Hash  (cost=1.02..1.02 rows=2 width=220)
        ->  Seq Scan on staff d  (cost=0.00..1.02 rows=2 width=220)
```

해당 실행계획은 Hash 조인과 Nested Loop 조인이 혼용되어 사용되고 있습니다. SQL Hint 를 이용하여 해당 SQL 문의 실행계획을 Hash 조인 및 테이블 풀 스캔으로 고정해보도록 합니다.

아래의 SQL 문을 살펴보도록 합니다.

[스크립트 3-8] HashJoin & SeqScan Hint 사용 후

```
/*+
   SeqScan(a)
   SeqScan(b)
   SeqScan(c)
   SeqScan(d)
   HashJoin(a b c d)
 */
select
         a.customer_id
       , b.rental_id
       , b.return_date
       , c.payment_id
       , c.payment_date
       , d.staff_id
       , d.first_name
       , d.last_name
 from
         public.customer a
       , public.rental b
       , public.payment c
       , public.staff d
where 1=1
  and a.customer_id between 388 and 400
  and a.customer_id = b.customer_id
  and b.rental_id = c.rental_id
  and c.staff_id = d.staff_id
;
```

모든 테이블을 Seq Scan 으로 처리하였고 Hash Join 으로 처리하였습니다.

아래의 실행계획을 살펴보도록 합니다.

```
Hash Join  (cost=397.01..712.64 rows=292 width=248)
  Hash Cond: (c.staff_id = d.staff_id)
  ->  Hash Join  (cost=395.96..707.58 rows=292 width=30)
        Hash Cond: (c.rental_id = b.rental_id)
        ->  Seq Scan on payment c  (cost=0.00..253.96 rows=14596
width=18)
        ->  Hash  (cost=391.95..391.95 rows=321 width=16)
              ->  Hash Join  (cost=18.13..391.95 rows=321 width=16)
                    Hash Cond: (b.customer_id = a.customer_id)
                    ->  Seq Scan on rental b  (cost=0.00..310.44
rows=16044 width=14)
                    ->  Hash  (cost=17.98..17.98 rows=12 width=4)
                          ->  Seq Scan on customer a  (cost=0.00..17.98
rows=12 width=4)
                                Filter: ((customer_id >= 388) AND
(customer_id <= 400))
  ->  Hash  (cost=1.02..1.02 rows=2 width=220)
        ->  Seq Scan on staff d  (cost=0.00..1.02 rows=2 width=220)
```

customer 테이블을 풀 스캔하여 rental 테이블과 해시 조인하고 다시 payment 테이블과 해시 조인한 후 staff 테이블과 해시조인한 것을 알 수 있습니다.

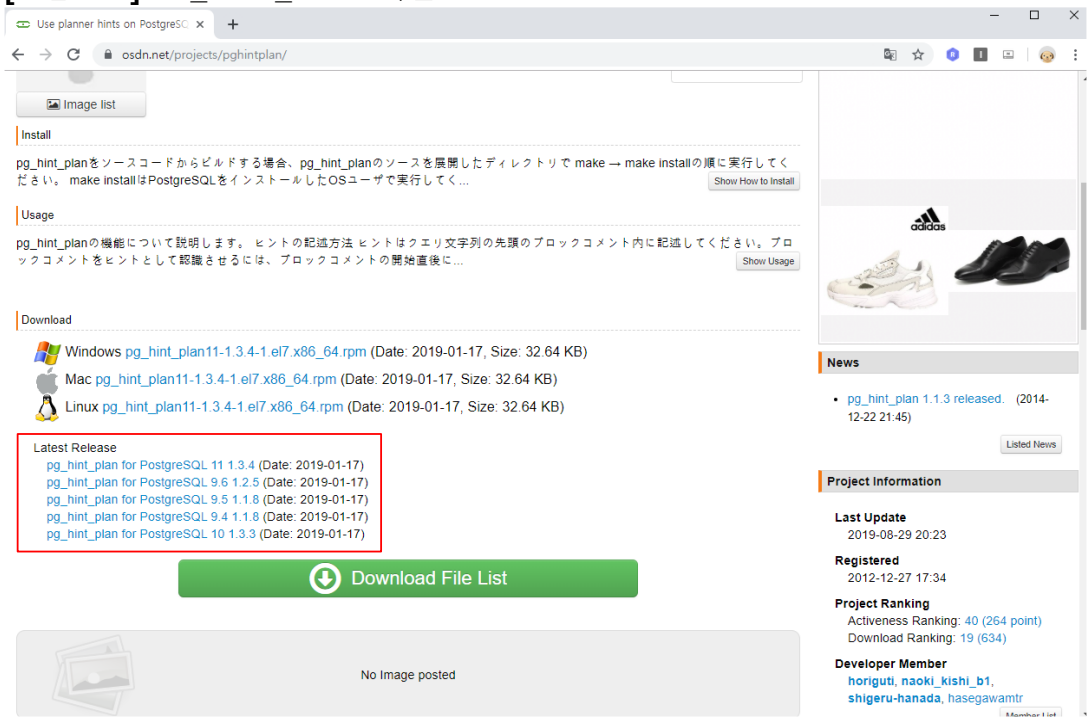
4. 기타지식

4.1. Linux 시스템에서 PG_HINT_PLAN 설치 방법

지금까지 우리는 AWS 의 Aurora 환경에서 PG_HINT_PLAN 을 사용하는 것을 학습하였습니다. 지금부터는 Linux 시스템에서 PG_HINT_PLAN 의 설치 방법에 대해서 알아보도록 합니다.

우선 https://osdn.net/projects/pghintplan/ 사이트에서 PG_HINT_PLAN 을 다운로드 합니다.

[그림 4-1] PG_HINT_PLAN 다운로드



자신이 사용하는 PostgreSQL 버전에 맞는 프로그램을 다운로드 합니다.

한글 사이트 주소에서 다운로드 받을 수도 있습니다.
주소는 https://ko.osdn.net/projects/pghintplan/releases/p15242 입니다.

다운로드 받은 파일을 아래와 같이 압축 및 Tar 를 해제합니다.

[스크립트 4-1] PG_HINT_PLAN 압축 해제 및 컴파일

```
$ tar xzvf pg_hint_plan-1.x.x.tar.gz
$ cd pg_hint_plan-1.x.x
$ make
$ su
```

```
# make install
```

아래와 같이 PG_HINT_PLAN 을 로딩 시킵니다.

[스크립트 4-2] PG_HINT_PLAN 라이브러리 로딩

```
postgres=# LOAD 'pg_hint_plan';
LOAD
postgres=#
```

해당 작업은 단발성으로 PG_HINT_PLAN 을 적용시키는 것이고 영구적으로 적용시키려면 postgresql.conf 에서 shared_preload_libraries 항목에 "pg_hint_plan" 을 추가해야합니다.

여기까지 작업 후 PostgreSQL 데이터베이스를 재 시작합니다. 재 시작 후 마지막으로 아래와 같은 명령을 실행합니다.

[스크립트 4-3] PG_HINT_PLAN Extension 추가

```
--익스텐션 추가
create extension pg_hint_plan;
```

Extension 추가가 완료되면 지금부터 SQL 문의 SQL Hint 를 이용할 수 있습니다.

4.2. 서브쿼리 실행 계획 제어

SQL 문에 서브 쿼리가 존재하면 옵티마이저는 그때부터 좀더 효율적인 실행 방안을 도출하게 됩니다. 즉 동일한 SQL 문이라도 해당 SQL 문을 실행하는 방법은 여러가지 일수 있기 때문입니다.

PostgreSQL 의 옵티마이저가 서브 쿼리를 제어하는 방식은 아래의 3 가지로 압축됩니다. (인라인 뷰의 쿼리 변환은 관점에서 제외)

[표 4-1] 서브쿼리 실행 방식 종류

방식	설명
서브 쿼리 Collapse	서브 쿼리 Collapse 는 서브 쿼리를 메인 쿼리에 병합해서 조인으로 유도하는 방법이다. 오라클 기준으로 Subquery Unnesting 이라고 할 수 있다.
세미 조인	세미 조인은 조건절의 조건을 부합하는 순간 해당 행에 대한 스캔을 멈추는 방식이다.
필터 방식	필터 방식은 메인 쿼리에서 반환한 row 마다 서브 쿼리를 수행하는 방식이다.

옵티마이저는 가장 먼저 서브 쿼리 Collapse 를 실행 계획으로 고려합니다. 또한 세미 조인으로 실행이 가능한 경우(ex. Exists 문 사용)에는 세미 조인 방식으로 실행됩니다. 즉 필터 방식은 옵티마이저 스스로 판단해서 실행할 가능성이 극히 낮은 것입니다. 이러한 이유로 오라클 에서도 널리 사용되는 튜닝 기법 중 하나

가 NO_UNNEST 힌트를 이용한 서브 쿼리 필터 처리 방식입니다. 반면에 PostgreSQL 은 별도의 PG_HINT_PLAN 에서 제공되는 SQL Hint 중 오라클의 NO_UNNEST 힌트와 완벽하게 매칭 되는 SQL 힌트는 존재하지 않습니다. 하지만 서브 쿼리 맨 끝에 "OFFSET 0"을 추가하는 방법으로 서브 쿼리의 실행 방식을 필터 방식으로 제어할수 있습니다.

그럼 지금부터 "OFFSET 0"으로 서브 쿼리를 필터 방식으로 처리하는 방법을 소개합니다. 우선 아래의 SQL 문을 실행하여 실습환경을 구축합니다.

[스크립트 4-4] 실습 환경 구성

```
--1. 테이블 생성
create table tb_emp (emp_no integer, sal integer, emp_nm char(100));
create table tb_ord (ord_no integer, emp_no integer, prdt_nm char(100));

--2. 임의의 데이터 생성
insert into tb_emp
select rand_num, rand_num, 'emp_nm'
  from generate_series(1, 10000) a(rand_num)
;

insert into tb_ord
select rand_num, rand_num, 'prdt_nm'
  from generate_series(1, 1000000) a(rand_num)
;

--3. 인덱스 생성
create index ix_tb_emp_01 on tb_emp(emp_no);
create index ix_tb_emp_02 on tb_emp(sal);
create index ix_tb_ord_01 on tb_ord(ord_no);
create index ix_tb_ord_02 on tb_ord(emp_no);

--4. 통계 정보 수집
analyse tb_emp;
analyse tb_ord;
```

위와 같이 tb_emp 테이블에는 1 만건 입력하고 tb_ord 테이블에는 100 만건을 입력합니다. 또한 tb_emp 테이블에는 emp_no 와 sal 컬럼에 대해서 각각 인덱스를 생성하고 tb_ord 테이블에는 ord_no 와 emp_no 컬럼에 대해서 각각 인덱스를 생성합니다. 인덱스 생성까지 완료되면 통계 정보를 수집하도록 합니다. 여기까지가 끝나면 실습 준비가 완료된 것입니다.

우선 이 상태에서 아래와 같은 SQL 문을 실행해보도록 합니다.

[스크립트 4-5] "OFFSET 0" 사용 전

```
explain(costs false, analyse, buffers)
select a.*
  from tb_emp a
 where a.sal between 1 and 100
   and exists (
               select 1
                 from tb_ord b
                where b.emp_no = a.emp_no
               )
```

위 SQL 문의 실행계획 및 내역은 아래와 같습니다.

```
Nested Loop Semi Join (actual time=0.023..0.274 rows=100 loops=1)
  Buffers: shared hit=404
  ->  Index Scan using ix_tb_emp_02 on tb_emp a (actual
time=0.014..0.039 rows=100 loops=1)
        Index Cond: ((sal >= 1) AND (sal <= 100))
        Buffers: shared hit=4
  ->  Index Only Scan using ix_tb_ord_02 on tb_ord b (actual
time=0.002..0.002 rows=1 loops=100)
        Index Cond: (emp_no = a.emp_no)
        Heap Fetches: 100
Buffers: shared hit=400
Planning time: 0.268 ms
Execution time: 0.305 ms
```

tb_emp 테이블을 Driving 테이블이 되고 tb_ord 테이블이 Driven 테이블이 되어 Nested Loop 조인으로 실행되었습니다. 즉 옵티마이저가 최적의 실행 방안을 찾아내어 실행된 것입니다. (실행 시간이 0.305 밀리 세컨드로 부하가 없었습니다.)

그럼 지금부터 아래와 같이 tb_ord 테이블의 주요 인덱스인 emp_no 컬럼의 인덱스를 삭제해 보도록 하겠습니다.

[스크립트 4-6] Driven 테이블 내 인덱스 삭제

```
drop index ix_tb_ord_02;
```

해당 인덱스는 Nested Loop 조인 시 가장 중요한 인덱스로써 해당 인덱스가 없다면 옵티마이저는 다른 실행 방안을 찾을 확률이 매우 높아집니다.

동일한 SQL 문을 다시 실행시켜보도록 합니다.

[스크립트 4-7] Driven 테이블 내 인덱스 삭제 후 실행

```
explain(costs false, analyse, buffers)
select a.*
  from tb_emp a
 where a.sal between 1 and 100
   and exists (
               select 1
                 from tb_ord b
                where b.emp_no = a.emp_no
               )
;
```

실행계획 및 내역을 살펴보면 아래와 같습니다.

```
Hash Semi Join (actual time=349.933..470.157 rows=100 loops=1)
  Hash Cond: (a.emp_no = b.emp_no)
  Buffers: shared hit=17249, temp read=2796 written=2766
  ->  Index Scan using ix_tb_emp_02 on tb_emp a (actual
time=0.014..0.040 rows=100 loops=1)
        Index Cond: ((sal >= 1) AND (sal <= 100))
        Buffers: shared hit=4
  ->  Hash (actual time=349.265..349.265 rows=1000000 loops=1)
        Buckets: 131072  Batches: 16  Memory Usage: 3227kB
```

```
        Buffers: shared hit=17242, temp written=2736
        ->  Seq Scan on tb_ord b (actual time=0.006..151.273 rows=1000000
loops=1)
Buffers: shared hit=17242
Planning time: 0.321 ms
Execution time: 470.445 ms
```

tb_ord 테이블의 emp_no 컬럼의 인덱스가 없어서 옵티마이저는 해시 세미 조인을 유도한 것을 알 수 있습니다. 수행 시간도 470.445 밀리 세컨드로 극단적으로 증가하였습니다. 하지만 옵티마이저 입장에서는 Nested Loop 조인으로 실행해서 Driven 집합을 100 번 풀 스캔 하는 것보단 이게 더 낫다고 판단해서 나름대로 최적의 성능으로 결과 집합을 돌려준 것입니다.

하지만 만약 해당 SQL 문을 필터 처리한다면 더 나은 성능이 나올 수도 있습니다. 아래와 같이 서브 쿼리 맨 끝에 "OFFSET 0"을 추가하여 SQL 문을 실행해 보도록 합니다.

[스크립트 4-8] "OFFSET 0" 추가 후 실행

```
explain(costs false, analyse, buffers)
select a.*
  from tb_emp a
 where a.sal between 1 and 100
   and exists (
               select 1
                 from tb_ord b
                where b.emp_no = a.emp_no offset 0
               )
;
```

서브 쿼리 맨 끝에 "offset 0"을 추가한 것을 주목합니다. 위 SQL 문의 실행 계획 및 내역은 아래와 같습니다.

```
Index Scan using ix_tb_emp_02 on tb_emp a (actual time=0.019..0.403 rows=100 loops=1)
  Index Cond: ((sal >= 1) AND (sal <= 100))
  Filter: (SubPlan 1)
  Buffers: shared hit=146
  SubPlan 1
    ->  Seq Scan on tb_ord b (actual time=0.003..0.003 rows=1 loops=100)
          Filter: (emp_no = a.emp_no)
          Rows Removed by Filter: 50
          Buffers: shared hit=142
Planning time: 0.126 ms
Execution time: 0.425 ms
```

"offset 0"을 추가하여 필터 방식으로 처리되었고 이전보다 성능이 극단적으로 빨라졌습니다. 수행 시간은 0.425 밀리 세컨드가 되었습니다. 즉 메인 쿼리의 결과 집합이 적고 서브 쿼리의 결과 집합이 테이블 앞쪽 블록에 있는 경우에는 필터 방식이 유리할 수 있습니다. 즉 "offset 0"을 지정하는 것만으로 필터 방식을 유도한 성능 개선에 성공한 것을 알 수 있습니다.

부가적으로 또 한가지 재밌는 사실은 "offset 0"을 이용해서 인라인 뷰의 뷰 머징

을 방지할 수 있습니다. 즉 인라인 뷰 내에 "offset 0"을 입력하면 Oracle 기준으로 NO_MERGE 힌트를 사용한 것과 같은 효과를 낼 수 있습니다. 해당 지식이 중요한 이유는 PG_HINT_PLAN 에서 제공하는 SQL Hint 중 NO_MERGE 와 완벽히 매칭 되는 것은 없기 때문입니다. 간단한 실습을 위해서 아래와 같은 SQL 문을 실행시켜보도록 합니다.

[스크립트 4-9] "OFFSET 0"을 이용한 View Merging 방지

```
explain(costs false, analyse, buffers)
select a.*, b.*
from tb_emp a
   , (select b.*
        from tb_ord b offset 0
     )  b
where a.sal between 1 and 100
and a.emp_no = b.emp_no
;
```

tb_ord 테이블이 b 인라인 뷰 안에 존재하며 해당 인라인 뷰 맨 끝에 "offset 0"을 준 것에 주목합니다. 이렇게 하면 오라클의 NO_MERGE 힌트를 사용한 것과 같은 효과를 누릴 수 있습니다. (뷰 머징을 방지 하는게 반드시 성능에 좋다는 것은 당연히 아닙니다.)

실행 계획 및 내역은 아래와 같습니다.

```
Hash Join (actual time=0.072..223.488 rows=100 loops=1)
  Hash Cond: (b.emp_no = a.emp_no)
  Buffers: shared hit=17246
  ->  Seq Scan on tb_ord b (actual time=0.004..109.439 rows=1000000
loops=1)
        Buffers: shared hit=17242
  ->  Hash (actual time=0.063..0.063 rows=100 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 22kB
        Buffers: shared hit=4
        ->  Index Scan using ix_tb_emp_02 on tb_emp a (actual
time=0.018..0.036 rows=100 loops=1)
              Index Cond: ((sal >= 1) AND (sal <= 100))
              Buffers: shared hit=4
Planning time: 0.162 ms
Execution time: 223.522 ms
```

tb_emp 테이블을 인덱스 스캔 한 후 Build Input 으로 생성한 후 tb_ord 테이블과 테이블 풀 스캔한 것을 알 수 있습니다. 즉 오라클 기준의 NO_MERGE 가 동작한 것입니다.

결론은 PostgreSQL 에서 "offset 0"을 사용하면 Oracle 기준의 서브쿼리 NO_UNNEST 및 인라인뷰의 NO_MERGE 를 유도할 수 있다는 것입니다.