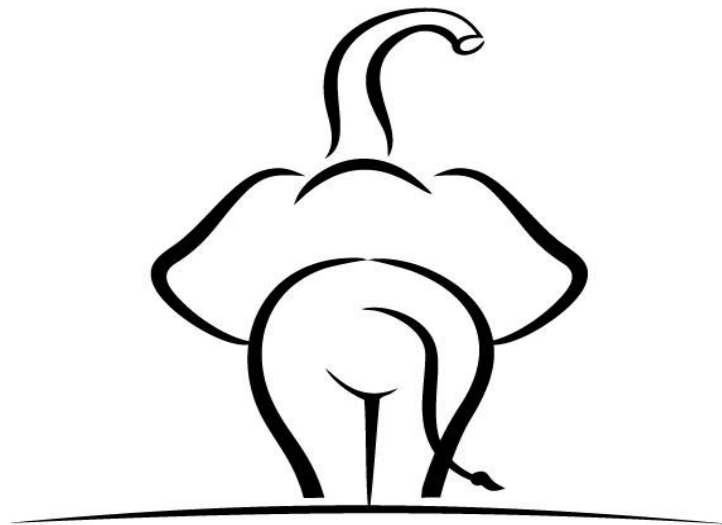


개발자를 위한 PostgreSQL Performance Tuning Workshop - 2017년 8월 교육자료

(부제: 1일 만에 PostgreSQL 튜닝 고수되기)

copyright 2017. 시연아카데미 all rights reserved.



저자의 허락없이 자료 사용이 가능합니다.
단, 출처를 꼭 밝혀주세요.

김 시연 - 시연아카데미 대표 (2016.4~)

주요 경력

- 오라클 성능 진단 및 튜닝 업무 (19년+)
- PostgreSQL 연구 (1년+)
- MySQL 연구 (2개월+)



직장 경력

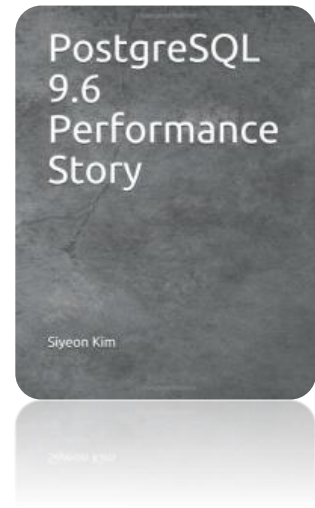
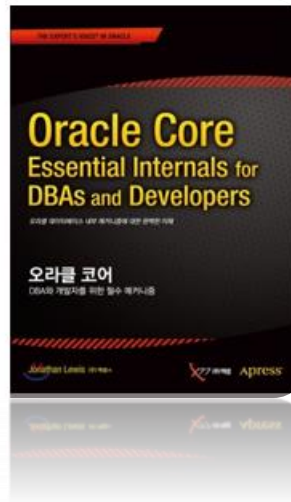
- (주) 엑셈 - 컨설팅 본부 본부장 (2001~2016)

컨설팅 수행 이력

- 2014년: 삼성전자 GSCM UI 성능 고도화 컨설팅
- 2013년: 삼성전자 GSCM 업그레이드 관련 안정화 및 EXADATA 성능 컨설팅
- 2012년: 삼성전자 GMES 2.0 성능 컨설팅
- 2011년: SKT 티맵 성능 컨설팅
- 2010년: 현대 글로비스 성능 컨설팅

단행본 출간 및 원서 번역

- 2017년 4월: PostgreSQL 9.6 Performance Story (영문버전)
- 2017년 2월: PostgreSQL 9.6 성능 이야기 (한글버전)
- 2012년 12월: Oracle 코어 번역
- 2008년 3월: Transaction Internals in 10gR2
- 2005년 2월: OWI를 활용한 오라클 진단 & 튜닝 번역



개발자들이 PostgreSQL 성능 튜닝을 위해서 꼭! 알아야할 핵심 내용을 짧은 시간 내에 전달하는 것을 목적으로 함

파트 1 - PostgreSQL 서버 아키텍처 (2시간)

1. 아키텍처 개요
2. Shared Buffer 동작원리의 이해 및 튜닝 방안 도출
3. MVCC와 Vacuum의 이해를 통한 성능 저하 예방 방안
4. PostgreSQL 파티션의 특징 및 튜닝 시 고려 사항

파트 2 - PostgreSQL 쿼리 옵티마이저 (4시간)

1. CBO 개요
2. 통계 정보
3. 성능 분석 도구인 Explain의 이해
2. Access 방식의 이해
3. Join 및 Query Rewrite
4. PG_HINT_PLAN을 이용한 쿼리 튜닝 방안
5. BRIN (Block Range Index) & Partial Index
6. Parallel 처리

Part-I.

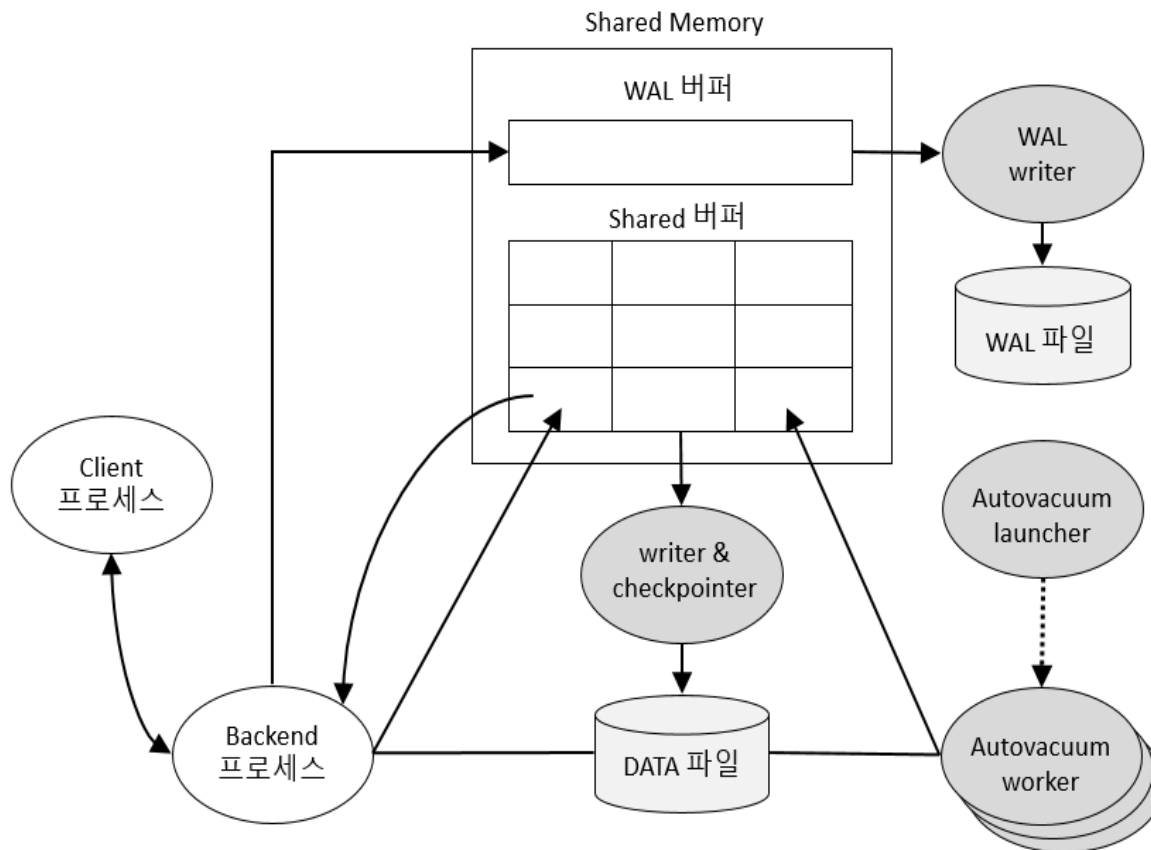
PostgreSQL 아키텍처

Architecture Overview

PostgreSQL 아키텍처

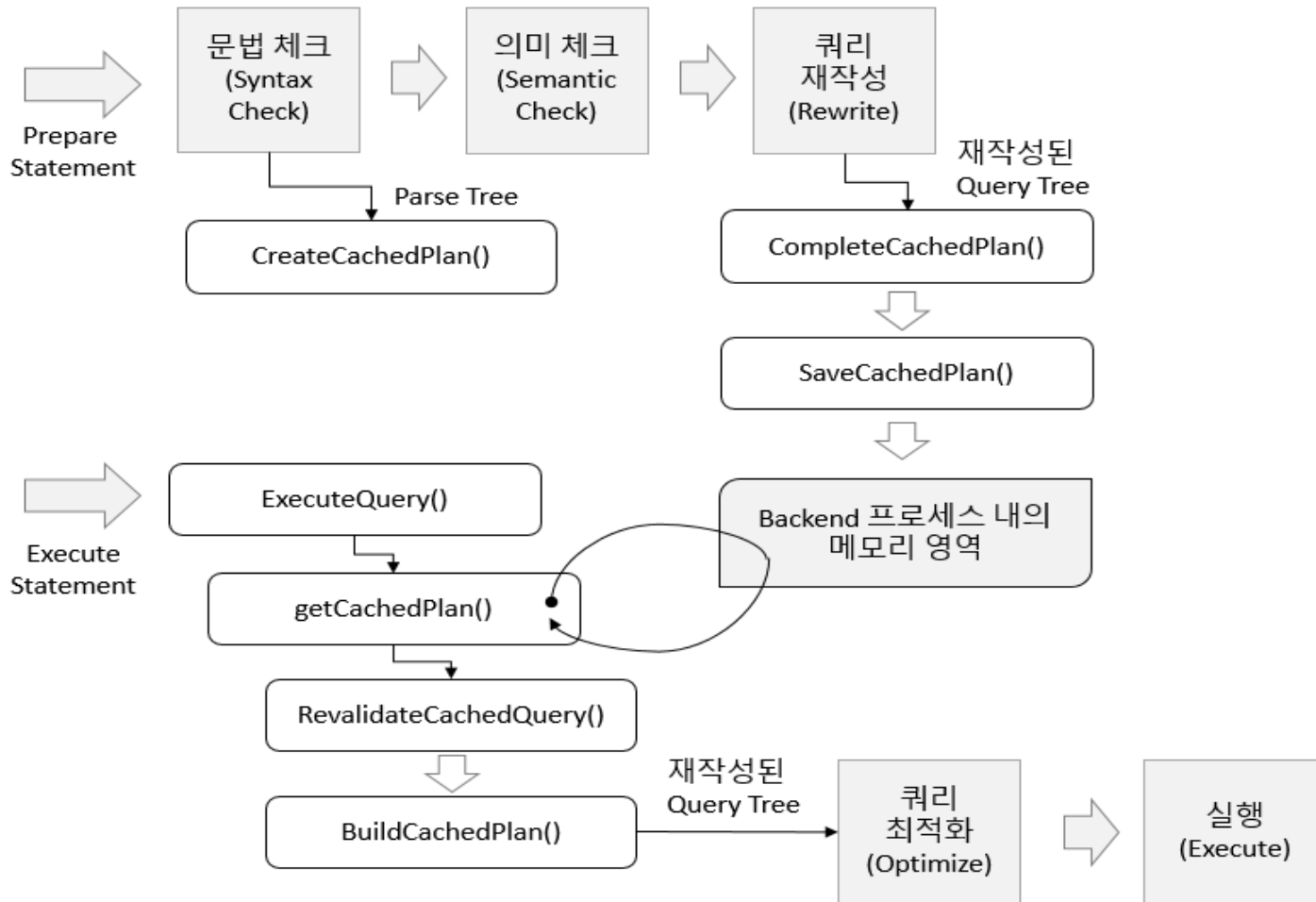
- PostgreSQL 아키텍처는 매우 단순하다.

- ✓ 공유 메모리, 매우 적은 수의 백그라운드 프로세스와 데이터파일로 구성된다.



여기서 잠깐! PostgreSQL은 SQL 공유를 위한 Pool이 없나?

- 없다. PostgreSQL은 세션 레벨의 Plan Caching 기능을 제공한다. 이를 이용한 Prepare Stmt 수행 구조도는 다음과 같다.



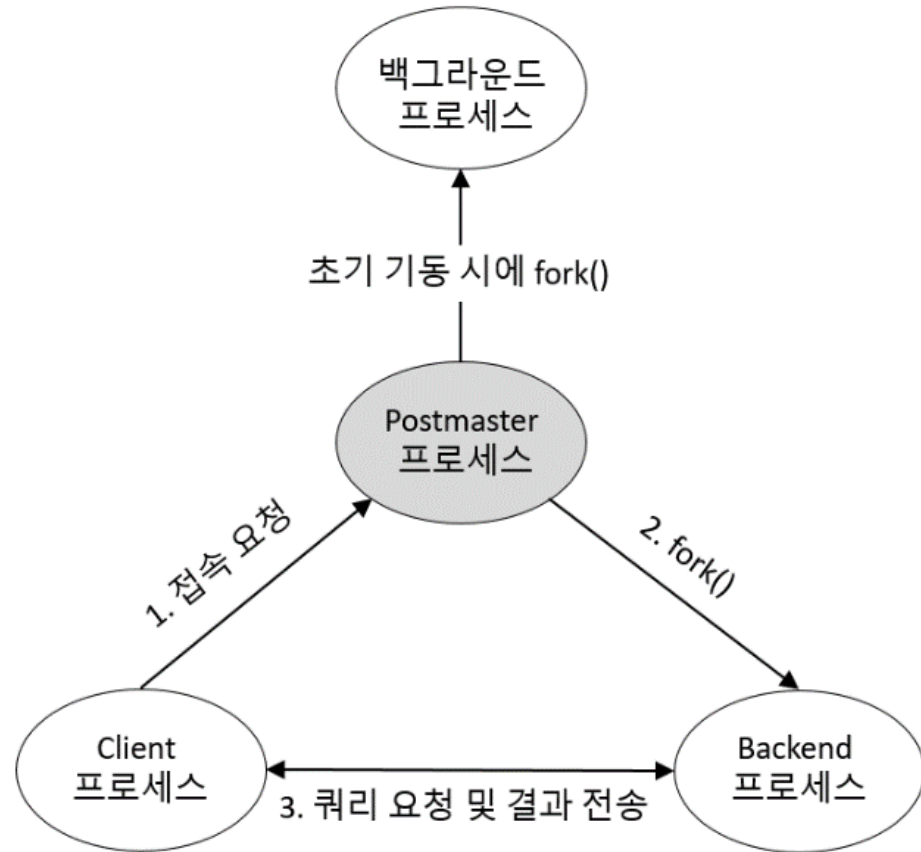
여기서 잠깐! Oracle Shared Pool vs. PostgreSQL Plan Caching

구분	Oracle	PostgreSQL
공유 레벨	Global	개별 세션 (초기 1회 수행 시는 Hard Parsing 발생)
관리 비용	매우 높음	없음
Bind Peeking	완벽하지 않음 (11g부터 Adaptive Cursor Sharing 제공)	완벽
통계 정보 변경 시의 영향도	높음 (10g부터 Cursor Invalidation 주기 조절 가능)	낮음
통계 정보 자동 갱신 주기	스케줄러로 조정	수집 기준 초과 시 자동 갱신

프로세스 유형

- PostgreSQL의 프로세스 유형은 4가지이다.

- ✓ Postmaster (Daemon) 프로세스
- ✓ Background 프로세스
- ✓ Backend 프로세스
- ✓ Client 프로세스



Backend 프로세스 메모리 구성

- Backend 프로세스 메모리와 관련된 파라미터 및 용도

파라미터	기본 설정값
work_mem	정렬 작업, Bitmap 작업, 해시 조인과 Merge 조인 작업 시에 사용되는 공간이다. 기본 설정값은 4 MiB이다.
maintenance_work_mem	Vacuum 및 CREATE INDEX 작업 시에 사용되는 공간이다. 기본 설정값은 64 MiB이다.
temp_buffers	Temporary 테이블을 저장하기 위한 공간이다. 기본 설정값은 8 MiB이다.

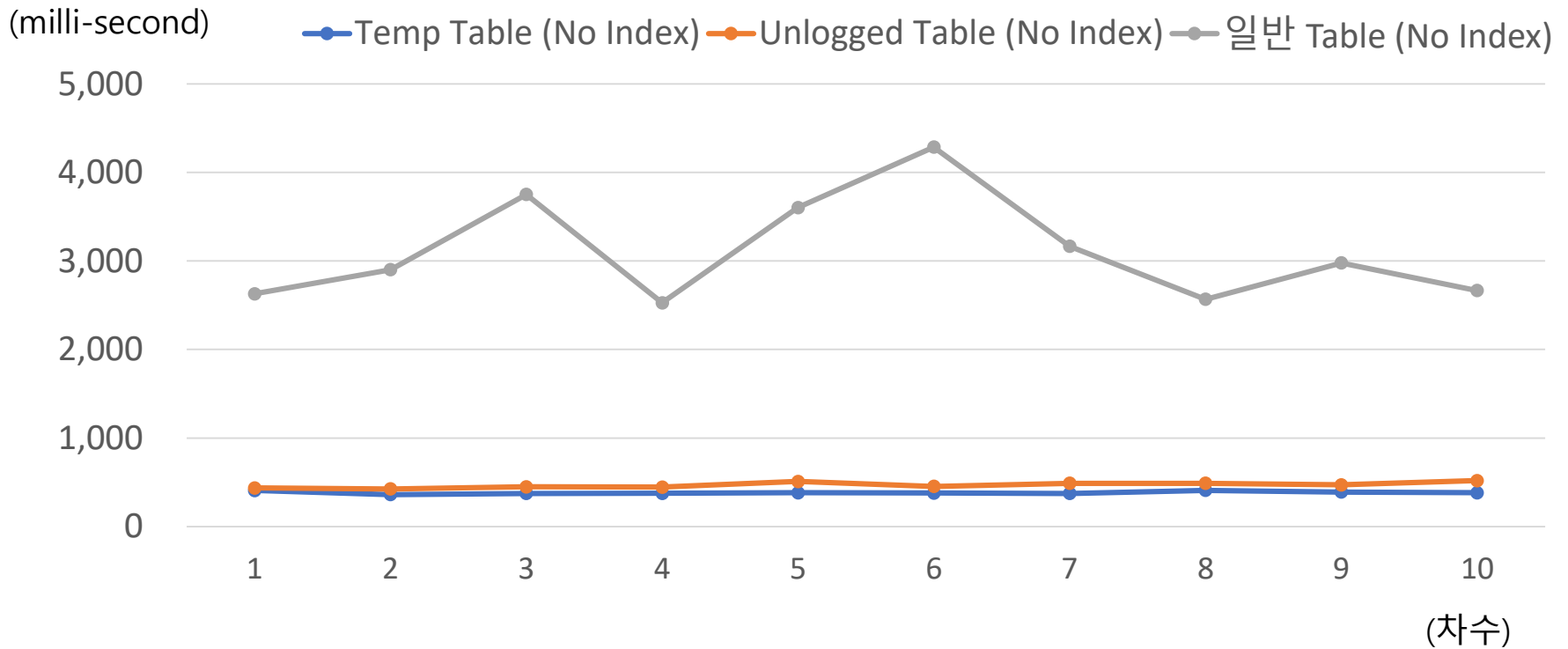
테이블 유형

구분	일반 테이블	UNLOGGED 테이블	Temporary 테이블
생성 방법	CREATE TABLE	CREATE UNLOGGED TABLE	CREATE TEMPORARY TABLE
WAL LOG 발생 여부	O	X	X
세션 간의 데이터 공유	O	O	X
COMMIT 후 데이터 저장 여부	O	O	Δ
DB 재 시작 후의 데이터 저장 여부	O	O	X
DB 장애 시의 데이터 복구 가능	O	X	X

✓ PostgreSQL의 UNLOGGED 테이블 (및 인덱스)은 WAL을 생성하지 않는다.

Temp 테이블 vs. UNLOGGED 테이블 vs. 일반 테이블 성능 비교

- 매회 100만 건 INSERT 수행
- 일반 테이블의 성능이 현저히 느린 이유는?
- 일반 테이블의 입력 성능이 3, 6회 시점에 약간의 스파이크가 발생한 이유는?



Shared Buffer & Tuning

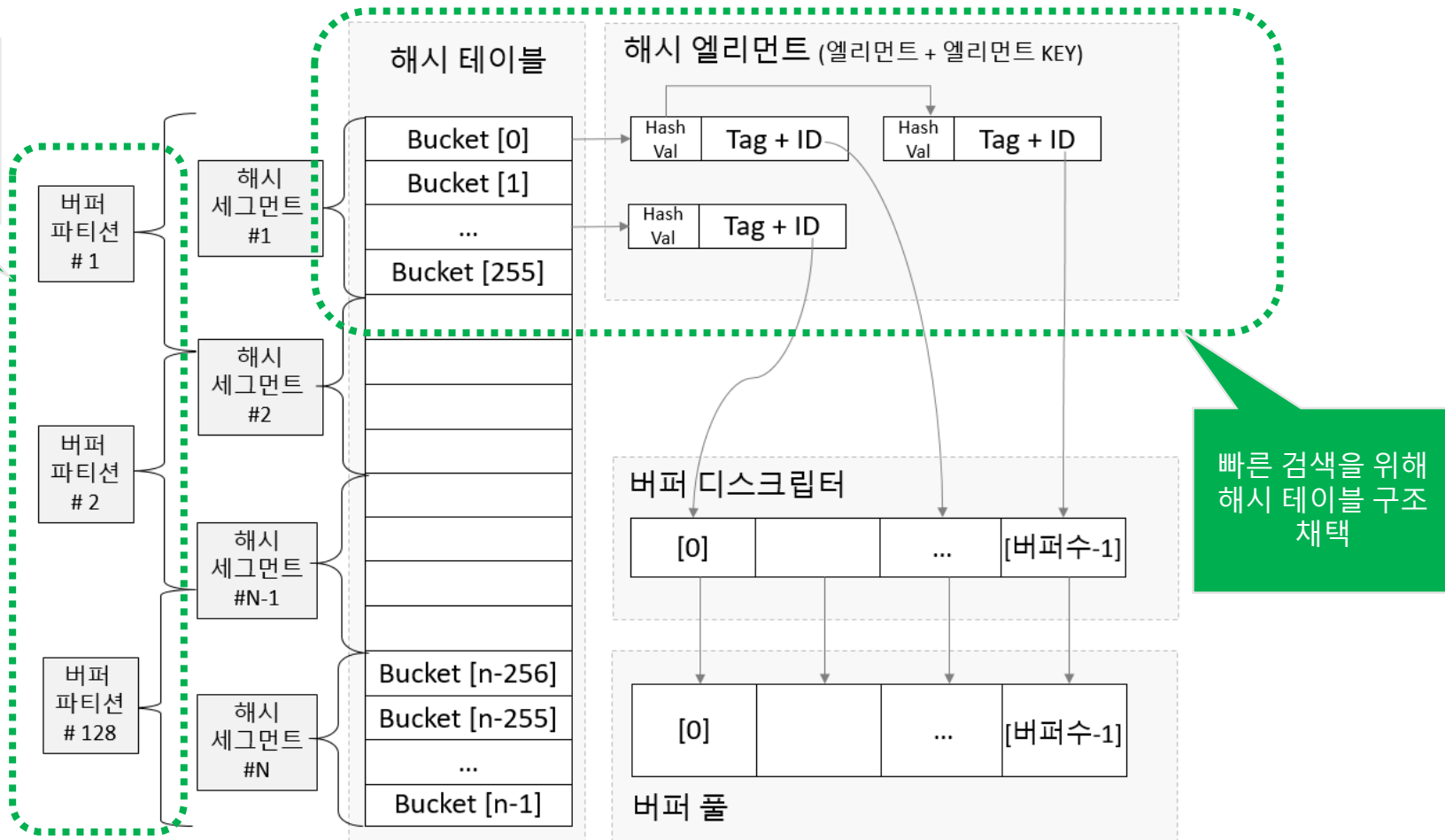
성능 향상을 위한 Shared Buffer의 목표

- Shared Buffer의 목적은 DISK IO를 최소화함으로써 IO 성능을 향상하는 것이다.
 - 1) 매우 큰 (수십, 수백 GB) 버퍼를 빠르게 액세스해야 한다.
 - 2) 많은 사용자가 동시에 접근할 때 경합을 최소화해야 한다.
 - 3) 자주 사용되는 블록은 최대한 오랫동안 버퍼 내에 있어야 한다.

Shared Buffer의 구조

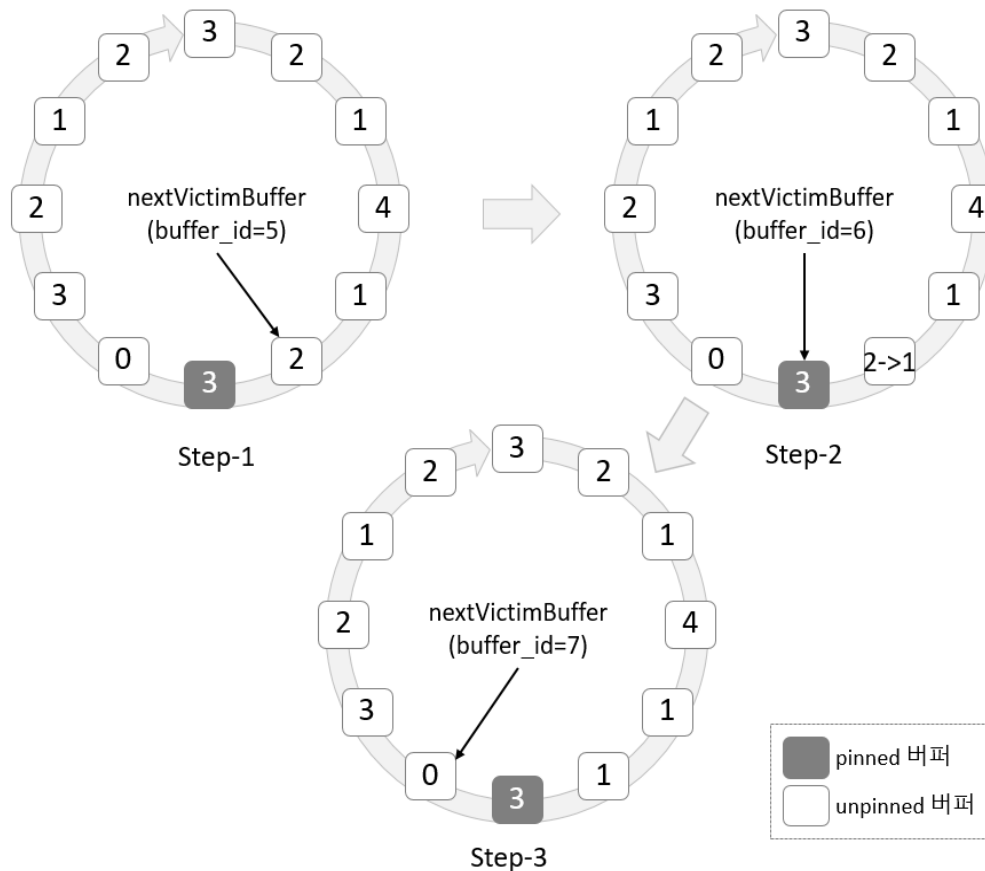
- Shared Buffer의 구조는 크게 1) 해시 테이블 2) 해시 엘리먼트 3) 버퍼 디스크립터 4) 버퍼 풀로 구성된다.

경합 완화를
위해 파티션
개수 증가
(128<-16,
9.5버전부터)



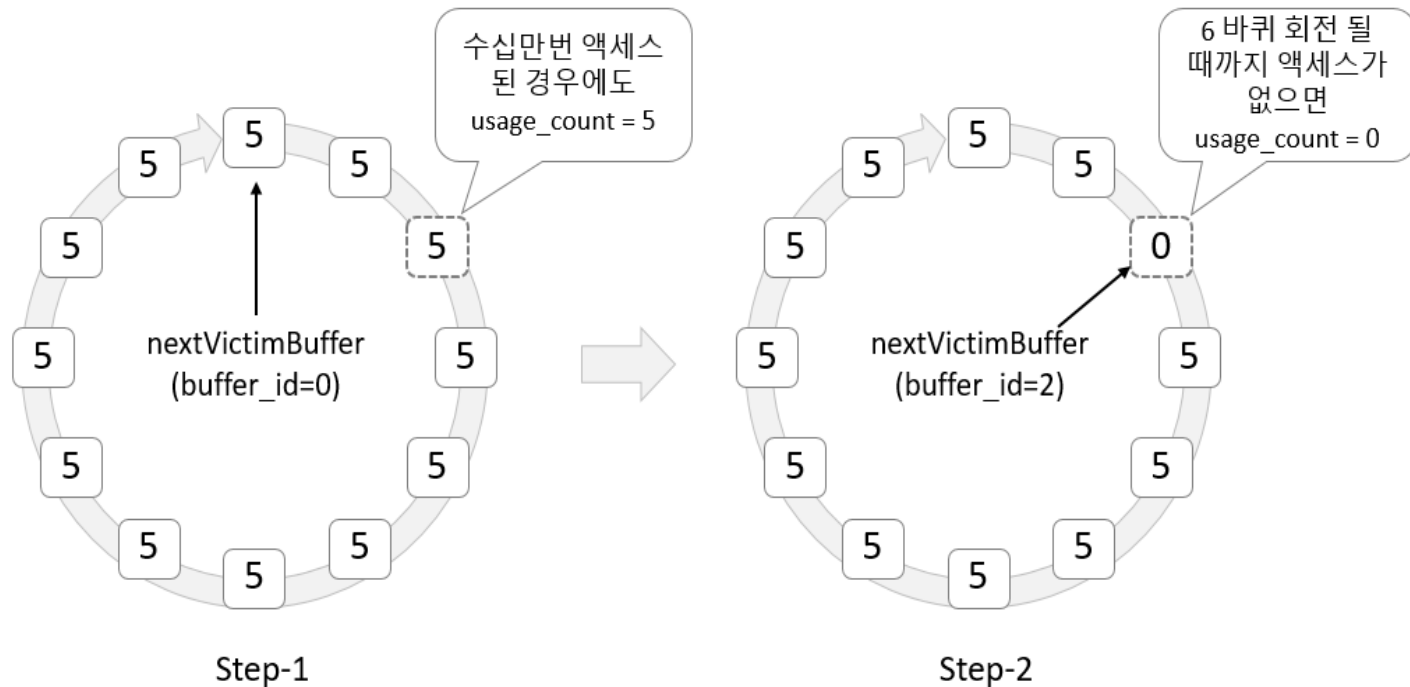
Buffer Replacement를 위한 Clock Sweep 알고리즘

- Clock Sweep 알고리즘은 덜 사용된 버퍼를 Victim 버퍼로 선정하는 알고리즘으로써 NFU (Not Frequently Used) 알고리즘의 일종이다.
- 버퍼가 액세스될 때마다 usage_count를 1씩 증가하며 최대 5 (BM_MAX_USAGE_COUNT) 까지 증가한다.



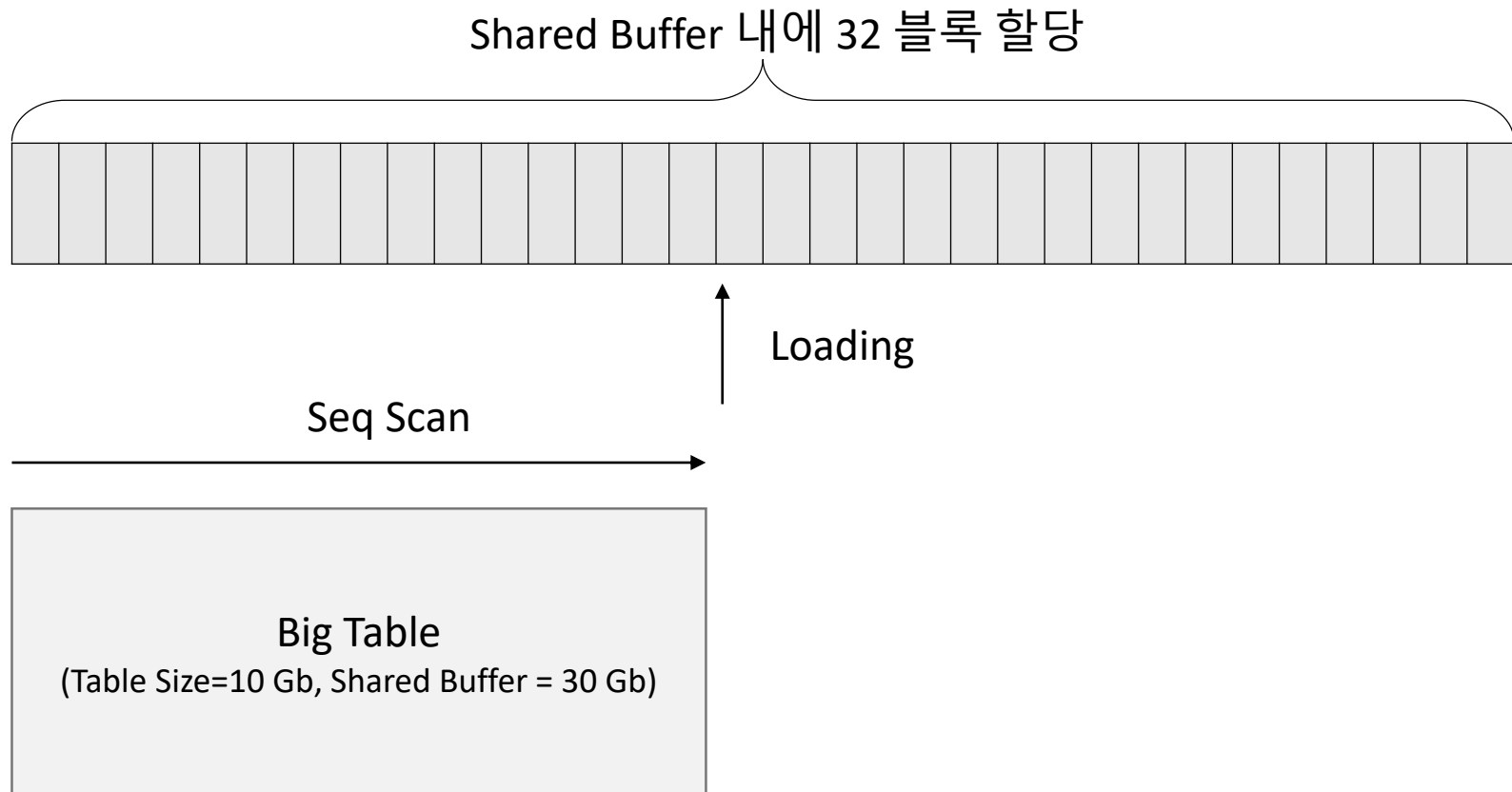
Clock Sweep 알고리즘의 공정성

- 짧은 시간에 아무리 많이 액세스되도 `usage_count`는 5를 넘지 않는다.
- Clock Sweep이 수행되면 `usage_count`는 1씩 감소한다.



Bulk Read로부터 Buffer를 보호하는 방법

- IO 유형을 4가지로 구분하고, 이때 Bulk Read인 경우에는 Ring Buffer를 사용하도록 한다.
- Bulk Read는 Shared Buffer의 $\frac{1}{4}$ 보다 큰 테이블에 대한 Seq Scan을 의미한다.
- Bulk Read를 위한 Ring Buffer의 크기는 32 블록이다.



pg_prewarm()익스텐션을 이용한 IO 튜닝 방안

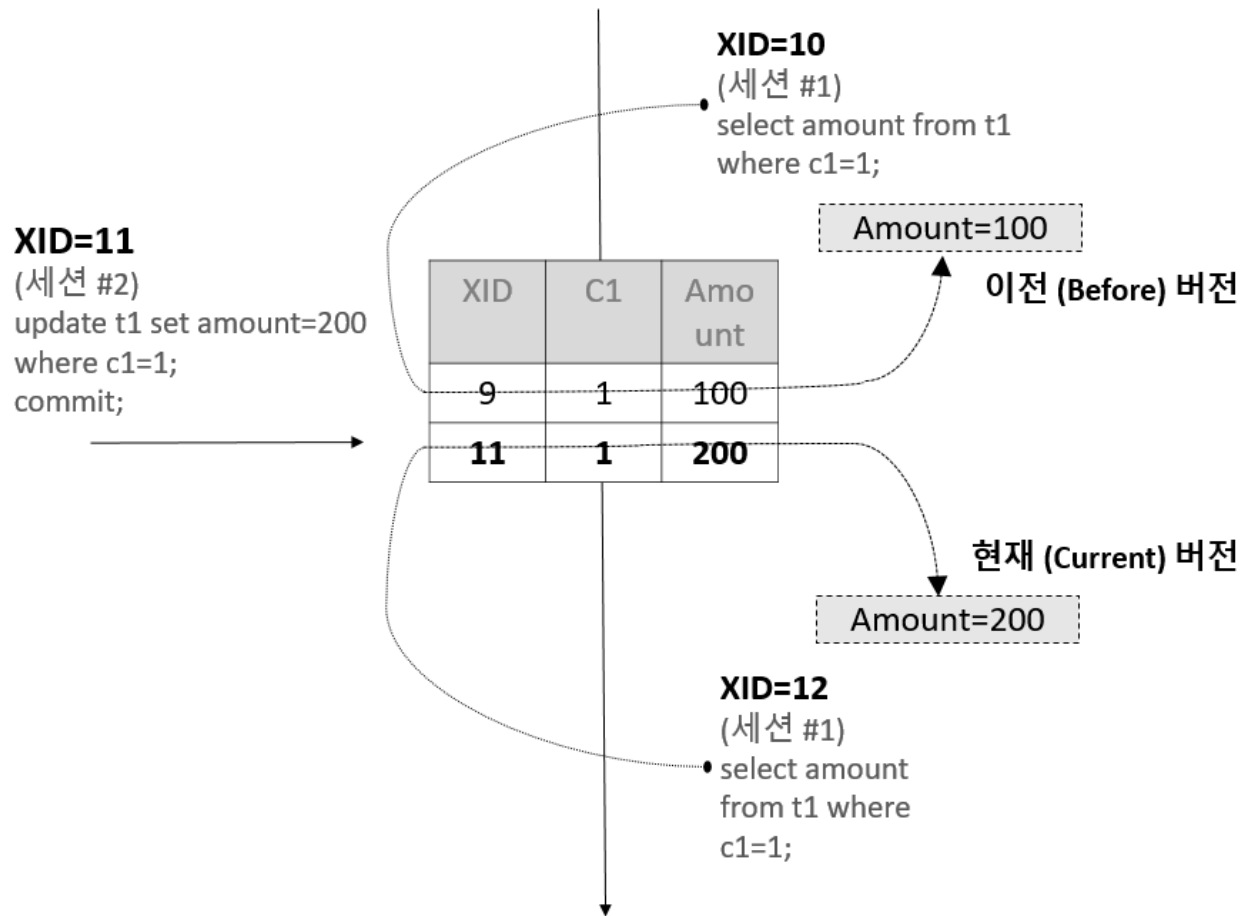
- pg_prewarm() 익스텐션을 이용하면 테이블과 인덱스를 Shared Buffer로 로딩할 수 있다.
- Shared Buffer의 ¼보다 큰 테이블도 로딩이 가능하다.
- 1회성 작업이다. 따라서 Batch 작업 전에 적절히 사용하면 IO 향상에 큰 효과를 낼 수 있다.

```
create extension pg_prewarm;  
  
select pg_prewarm('t1');
```

MVCC & Vacuum

MVCC란?

- MVCC (Multi-Version Concurrency Control)는 쿼리 수행 시점의 데이터를 제공하는 기법이다.
- 이를 위해서는 변경 후의 현재 데이터 (Current Data) 뿐 아니라, 변경 전의 이전 데이터 (Before Data)를 읽을 수 있어야 한다.



PostgreSQL MVCC의 특징

- 특징-1. 이전 버전의 데이터를 테이블 블록 내에 저장한다.

- 1) 이 특징은 MVCC를 매우 단순하게 구현할 수 있다는 장점이 있다.
- 2) 하지만 '이전 데이터'를 블록 내에 저장함으로써 테이블의 공간 사용 효율이 떨어진다는 단점이 있다.
- 3) 이 단점을 해결하기 위해서 Vacuum이 필요하다.
- 4) Vacuum을 수행하면 불필요한 데이터가 삭제된다.
- 5) Vacuum Full을 수행하면 테이블이 재생성 된다.

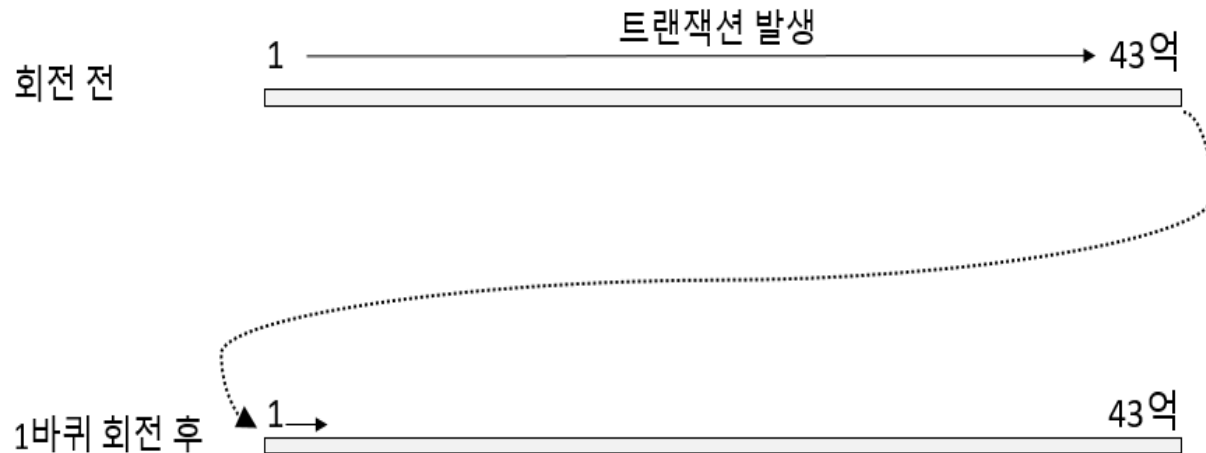
- 특징-2. 레코드 별로 트랜잭션 ID (XID)를 관리한다.

- 1) 테이블 블록 내에 '이전 데이터'를 저장하므로 레코드마다 트랜잭션 ID가 필요하다.
- 2) 이때, 레코드마다 트랜잭션 ID(XID)를 저장하므로 XID의 크기를 최소화할 필요가 있다.
- 3) PostgreSQL은 XID를 4바이트로 관리한다.
- 4) 이때문에 Vacuum이 필요하다.

정리하면, Vacuum은 PostgreSQL의 MVCC의 특징때문에 필요한 오퍼레이션이다.

트랜잭션 ID (XID)를 4바이트로 관리함에 따른 문제점

- 43억을 모두 소진한 후에는 다시 1부터 시작할 수밖에 없다.
- 1,000 TPS가 발생하는 시스템이라면, 50일 후에 43억을 모두 소진하게 된다.
(49.7일 = 43억 / (86,400초 * 1,000 TPS))



Wraparound 후에 XID 비교는 어떻게 할까?

- Wraparound 이전 시점

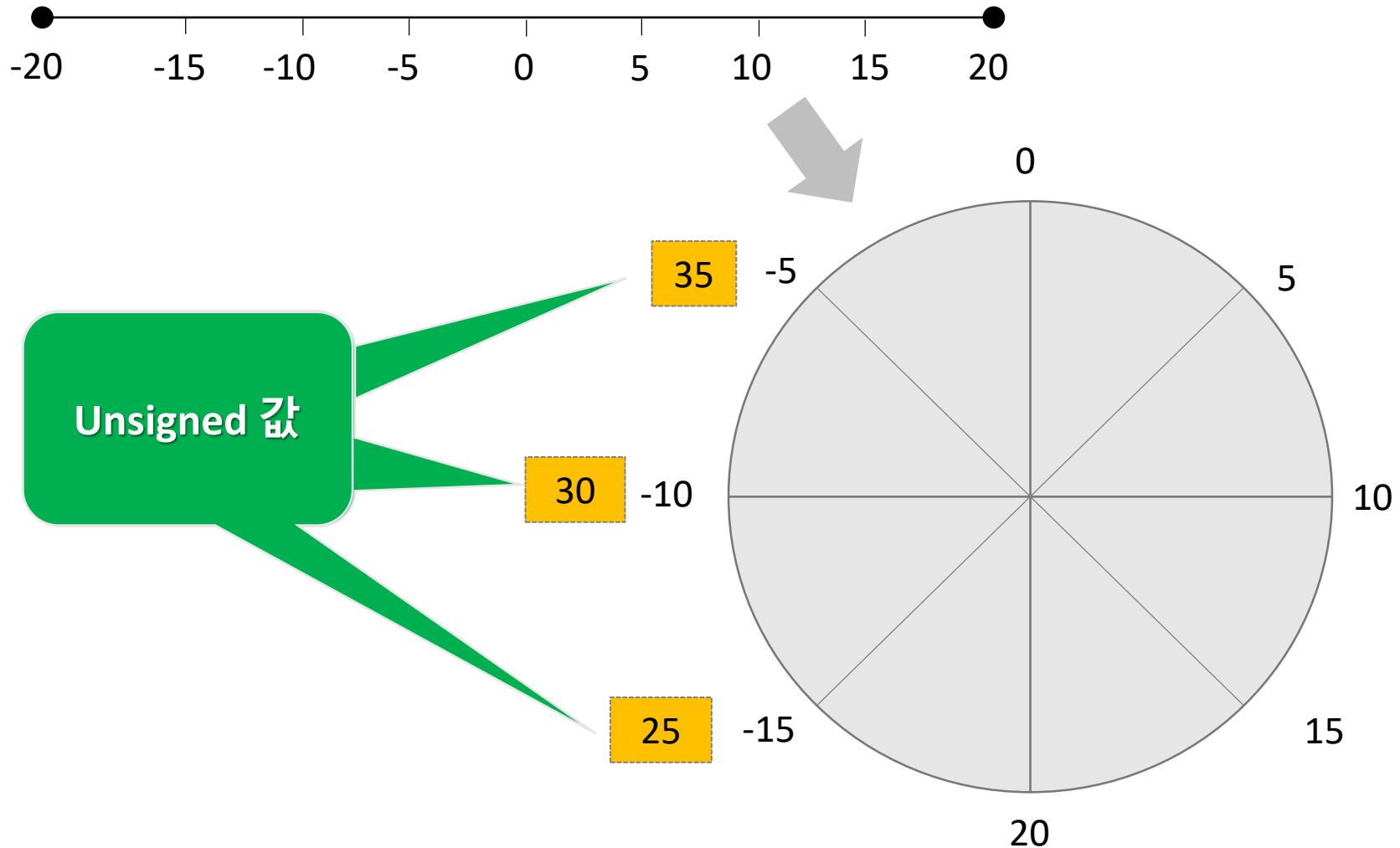
XID 43 억 > XID 42억 (당연한 수식)

- Wraparound 이후 시점

XID 3 > XID 43억 (어떻게 이런 수식이?)

PostgreSQL은 XID 비교를 위해 Modulo 연산을 사용한다.

- 직선을 구부려서 원을 만든다고 생각해보자. (편의상 0~40억으로 생각하자)

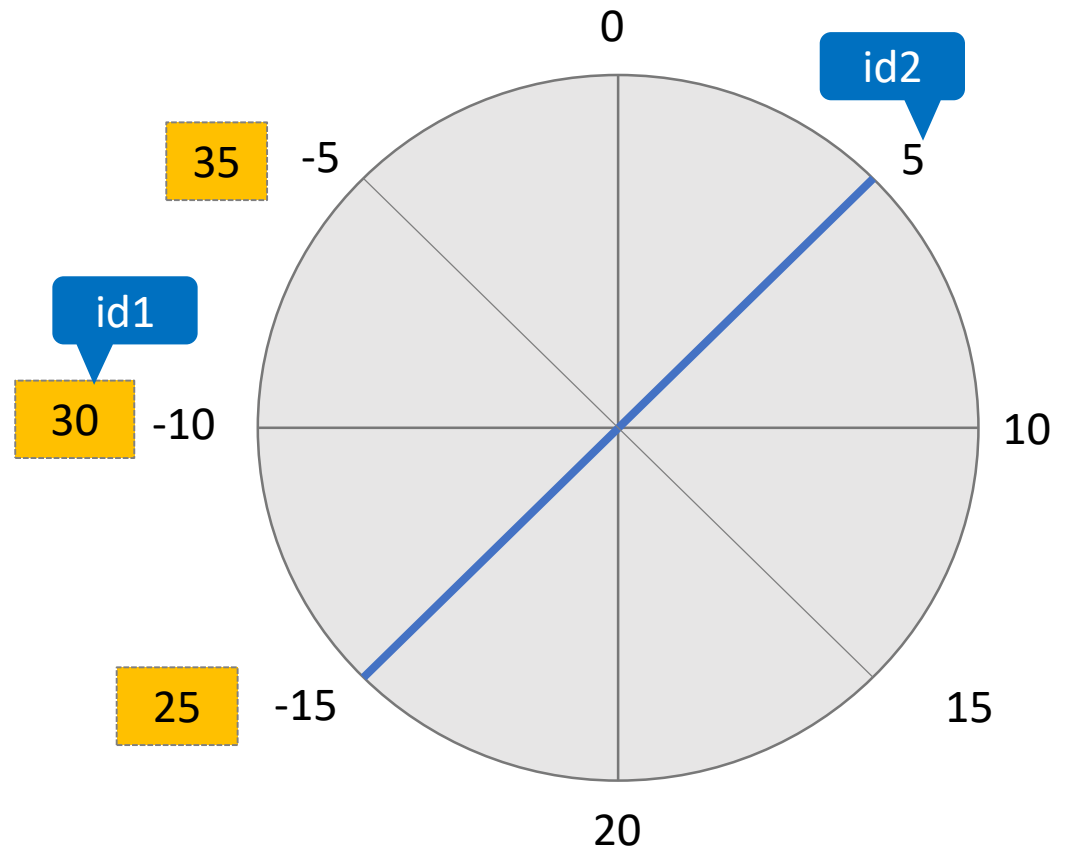


예제 #1

- 30와 5를 비교해보자. 어떤 값이 더 큰 값인가?
- 공식을 적용하면 5가 더 큰 값이다.

```
diff
= (int32) (id1 - id2)
= (int32) id1 - (int32) id2
= -10 - 5
= -15
```

즉, diff 결과가 마이너스이므로
id2가 더 큰 값이다.

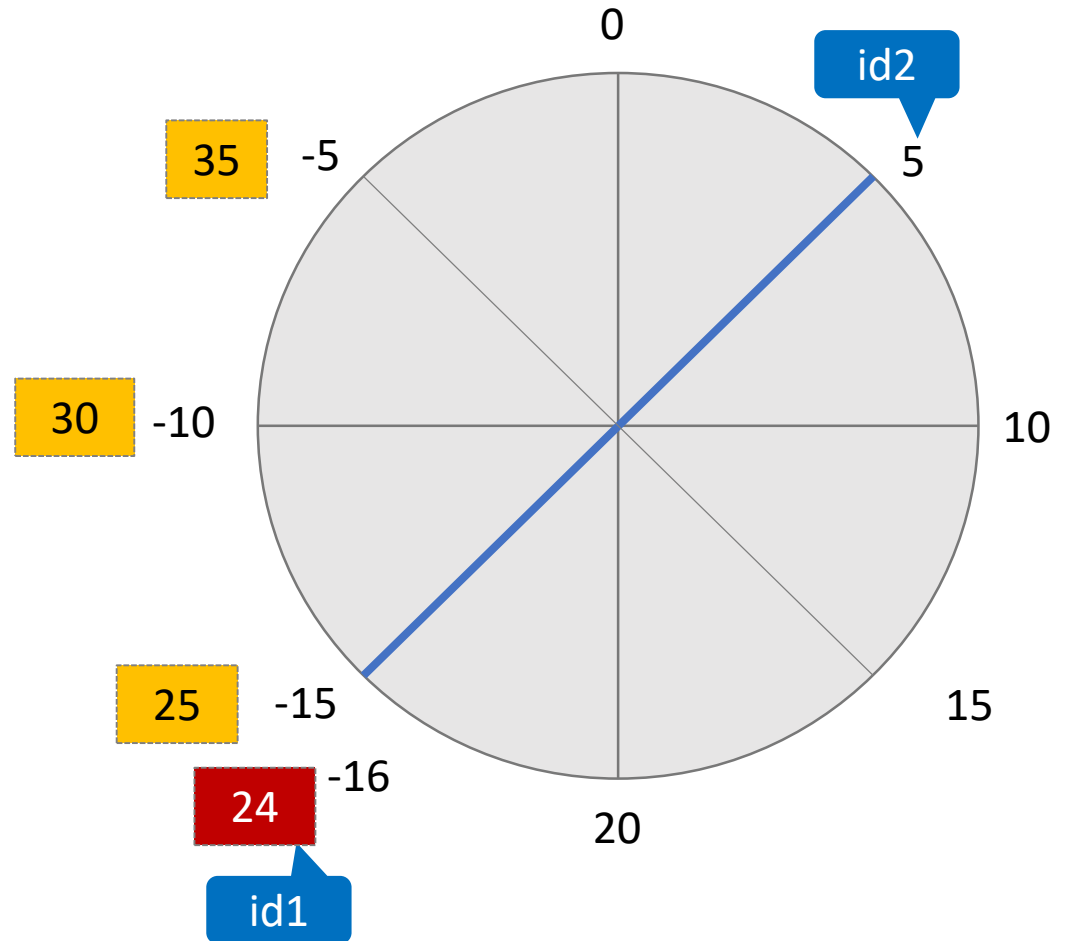


예제 #2

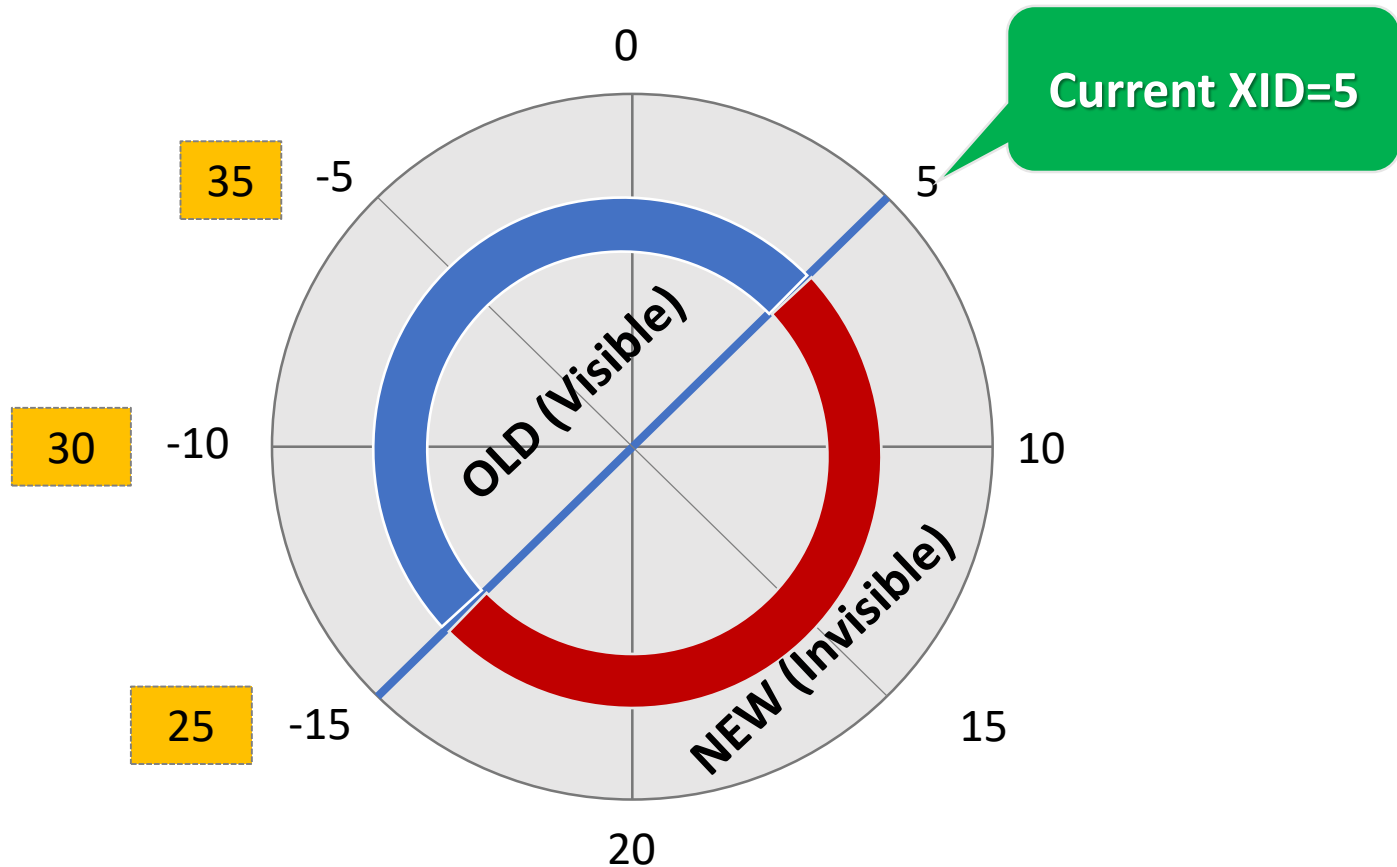
- 24와 5를 비교해보자. 어떤 값이 더 큰 값인가?
- 공식을 적용하면 24가 더 큰 값이다.

```
diff
= (int32) (id1 - id2)
= (int32) id1 - (int32) id2
= -16 - 5
= -21
→ -21은 -20~20 범위를 벗어나므로
1로 변환된다.
```

즉, diff 결과가 0보다 크므로
id1이 더 큰 값이다.

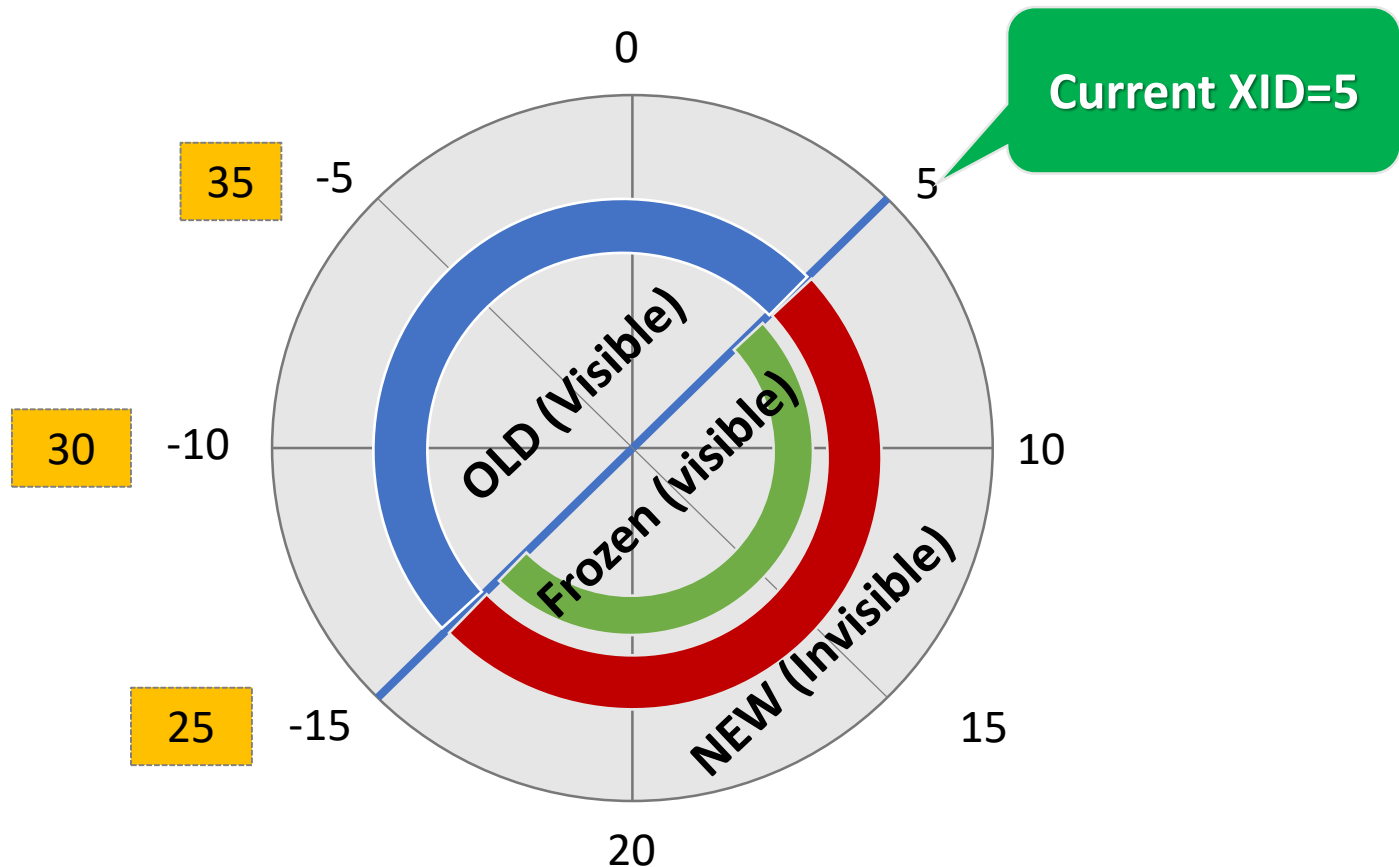


즉, 현재 XID 기준으로 절반은 OLD 데이터, 절반은 NEW 데이터이다.



한바퀴가 돌기 전에, NEW 데이터들은 FROZEN 돼야한다.

- Frozen된 데이터들은 항상 Visible 하다.
- Frozen을 위해, 9.3까지는 XMIN을 2로 변경했고, 9.4부터는 t_infomask 10번째 비트를 1로 변경한다.



Vacuum의 목적

- 공간 재활용

- 1) 오래된 이전 버전 레코드 삭제 작업을 통한 공간 확보 (Vacuum)
- 2) 오래된 이전 버전 레코드 삭제 작업 후에 공간 압축 (Vacuum Full)

필수
작업은
아님

- XID Frozen

- ✓ Vacuum 수행 시에 필요한 레코드에 대해서 수행됨
- ✓ Anti-Wraparound Vacuum 이라고 함

필수 작업

Vacuum과 Lock

- Vacuum은 DML과 호환된다.

세션#1

```
postgres=# begin;  
postgres=# lock table t1 in row exclusive mode;
```

세션#2

```
postgres=# vacuum t1;  
VACUUM
```


Vacuum Full과 Lock

- Vacuum Full은 DML 뿐 아니라 SELECT와도 호환되지 않는다. (PG_REPACK 익스텐션 고려)

세션#1

```
postgres=# begin;  
postgres=# lock table t1 in access share mode;
```

세션#2

```
postgres=# vacuum full t1;  
-- 락 대기
```

락 모니터링

```
select pid, wait_event_type||'-'||wait_event wait,  
       pg_blocking_pids(pid) holder, query  
from   pg_stat_activity;
```

pid	wait	holder	query
19309		{}	lock table t1 in access share mode;
20674	Lock-relation	{19309}	vacuum full t1;

Autovacuum

- Autovacuum이 하는 일

1) 데이터 변경에 따른 자동 통계 수집

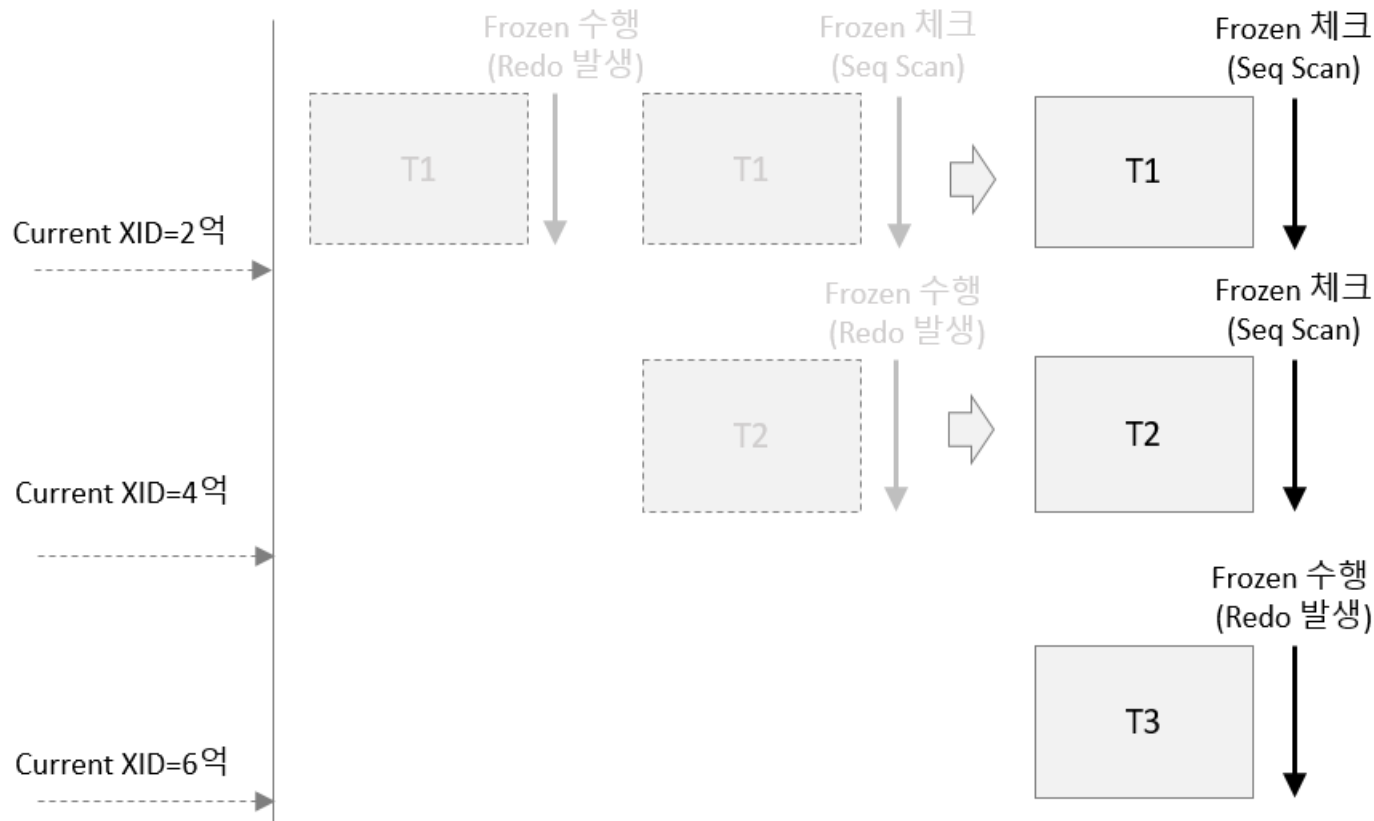
파라미터 명	설명	설정값
autovacuum	autovacuum 프로세스 사용 여부	on
autovacuum_analyze_scale_factor	테이블 내의 레코드 변경 비율	0.1
autovacuum_analyze_threshold	최소 변경 레코드 수	50

2) XID 증가에 따른 Anti-Wraparound Vacuum 수행

파라미터 명	설명	설정값
autovacuum_freeze_max_age	설정 값 이상의 나이(Age)인 테이블에 대한 Vacuum 및 XID Frozen 작업 수행	200,000,000

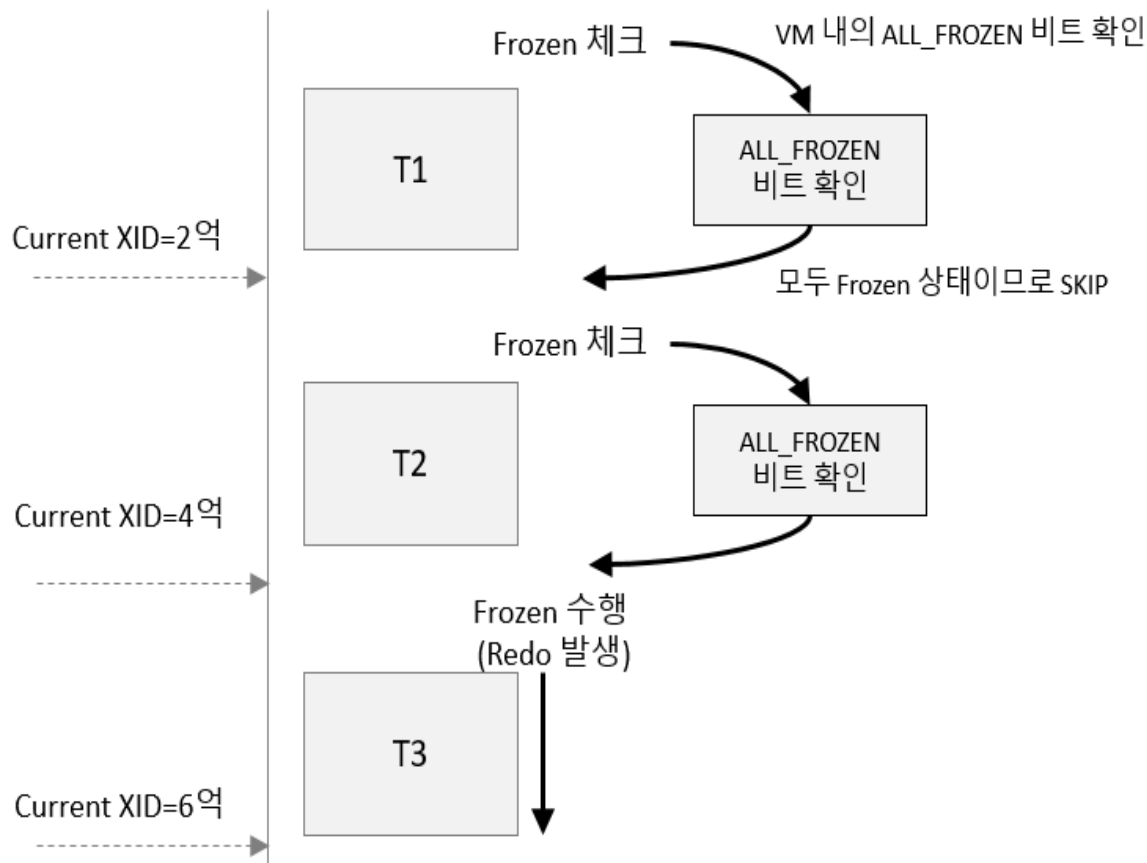
9.6 이전까지 Anti-Wraparound Vacuum의 문제점

- `autovacuum_freeze_max_age` = 2억 (기본 설정값)
- `vacuum_freeze_min_age` = 0 (기본 설정값=5천만)
- 아래와 같이 반복적인 테이블 Scan이 발생한다.
- 업무 시간에 이러한 Scan 작업이 발생하면 IO 경합으로 인한 성능 지연이 발생한다.



9.6 버전부터 Anti-Wraparound Vacuum 문제가 해결됨

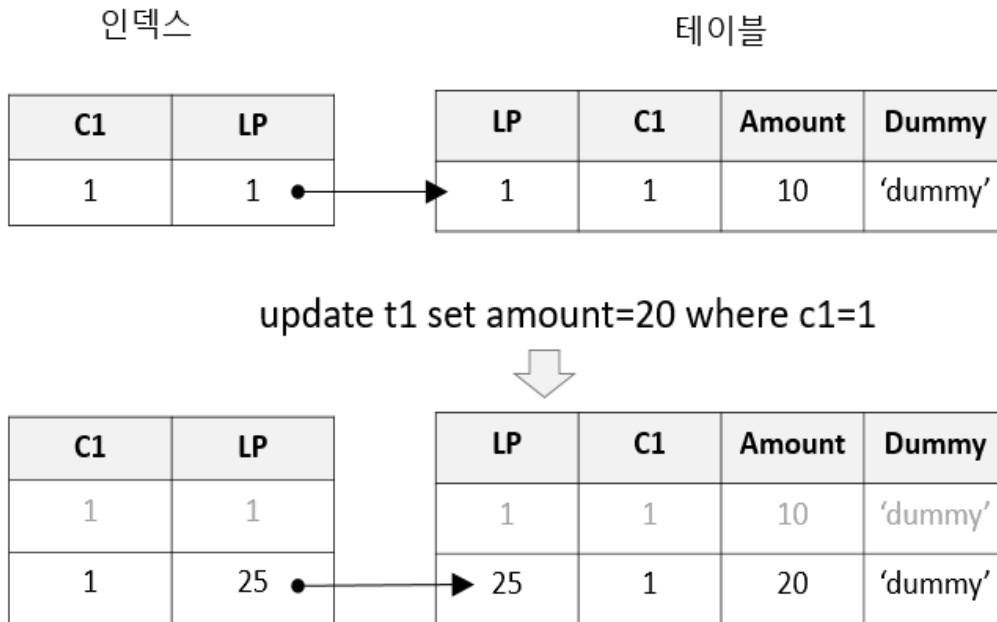
- Visibility Map에 ALL_FROZEN 비트를 추가함
- Visibility Map은 블록 당 2 비트로 구성됨 (1비트는 ALL_VISIBLE, 1비트는 ALL_FROZEN)
- 블록 내의 모든 레코드가 Frozen 됐으면 ALL_FROZEN 비트는 1로 설정됨
- 따라서 다음 Frozen 작업 시에 해당 블록들은 SKIP됨



HOT (Heap Only Tuple)

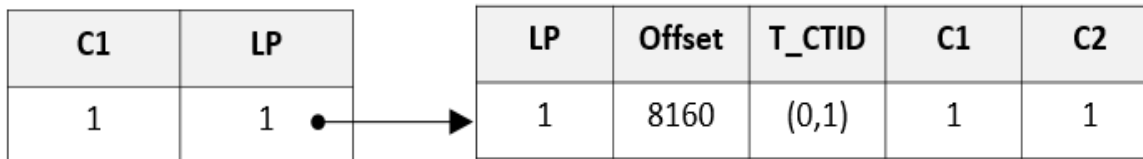
- HOT는 PostgreSQL MVCC 모델의 약점을 극복하기 위한 튜닝 기법이다.
- HOT는 '테이블에만 존재하는 레코드'란 의미이다.
- 풀어서 말하면, 인덱스에는 존재하지 않고 테이블에만 존재하는 레코드라는 뜻이다.
- HOT는 변경 후의 레코드가 동일 블록에 저장될 때만 동작한다.
- 따라서 변경이 빈번한 테이블은 반드시 FILLFACTOR를 작게 설정해야 한다.

HOT 도입 이전의 문제점

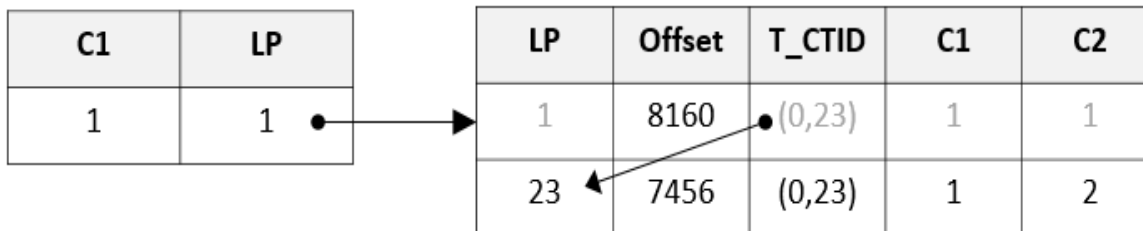


HOT (Heap Only Tuple)

1차 변경 후 (HOT 적용)

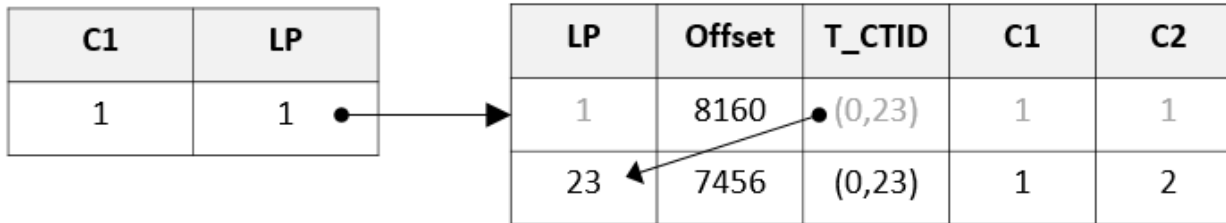


update t1 set c2=2 where c1=1

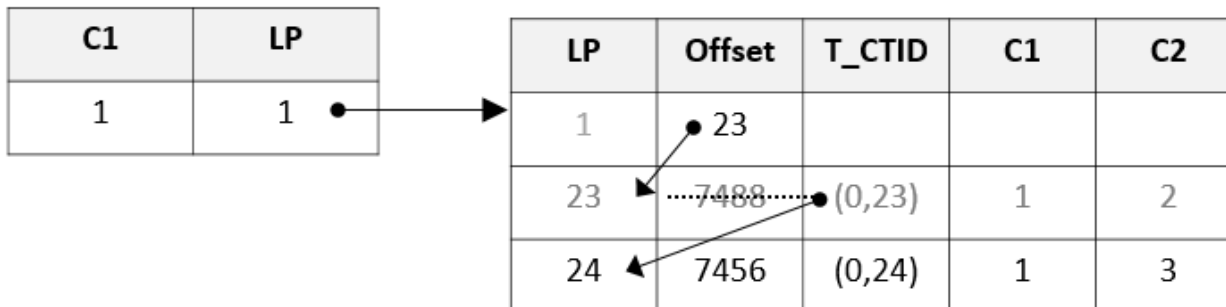


HOT (Heap Only Tuple)

2차 변경 후 (HOT 적용)



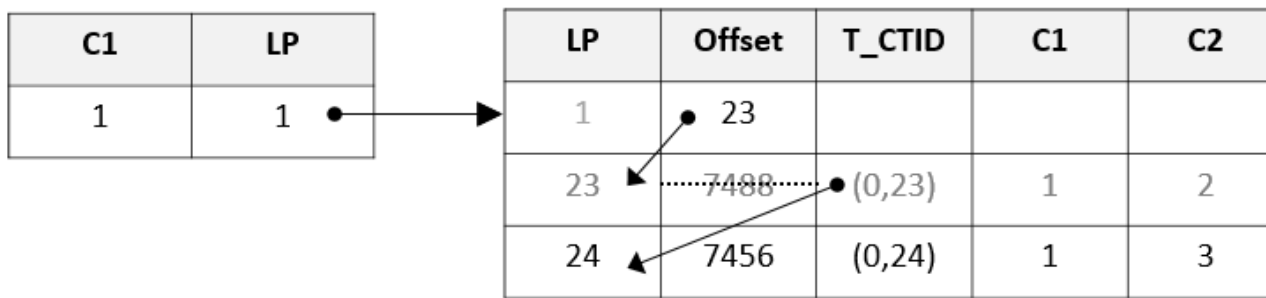
update t1 set c2=3 where c1=1



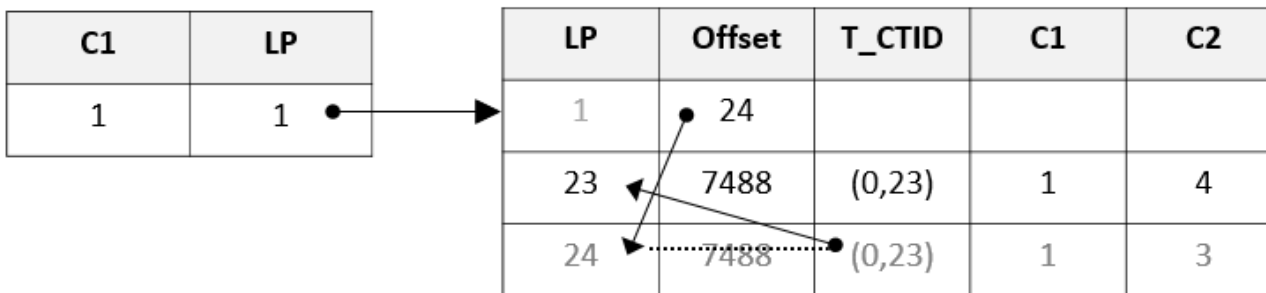
HOT (Heap Only Tuple)

3차 변경 후 (HOT 적용)

- HOT Chain Pruning을 통해 Chain 길이를 짧게 유지함



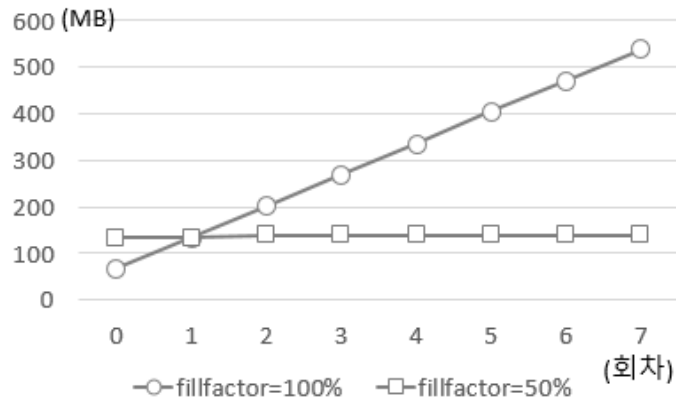
update t1 set c2=4 where c1=1



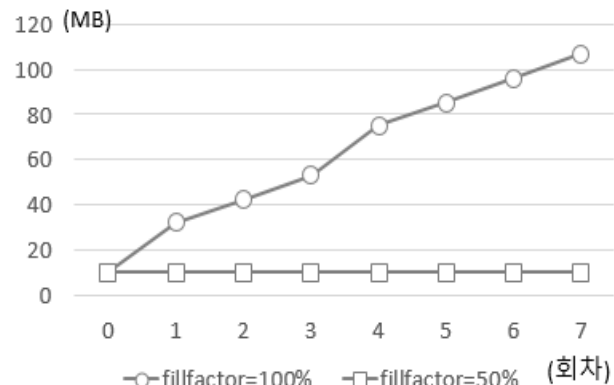
HOT (Heap Only Tuple)

- 변경이 빈번한 경우, FILLFACTOR가 100%(기본 설정값)인 경우와 50%인 경우의 비교

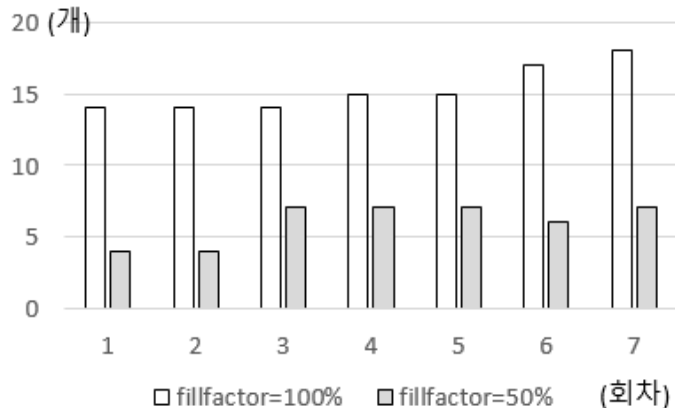
테이블 크기



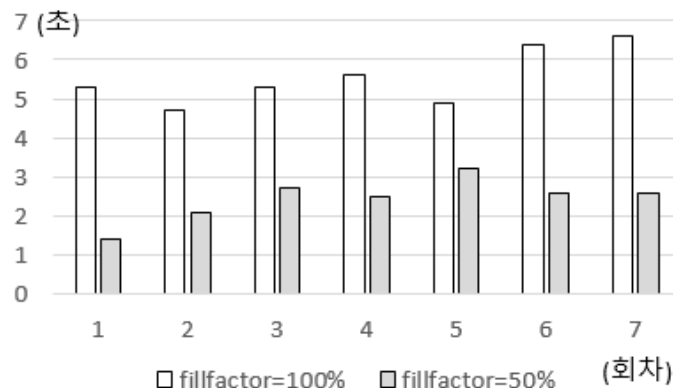
인덱스 크기



WAL 로그 개수



UPDATE 수행 속도



HOT (Heap Only Tuple)

- FILLFACTOR 적용 방법

```
create table t1 (c1 integer, c2 integer) WITH (fillfactor=50);
```

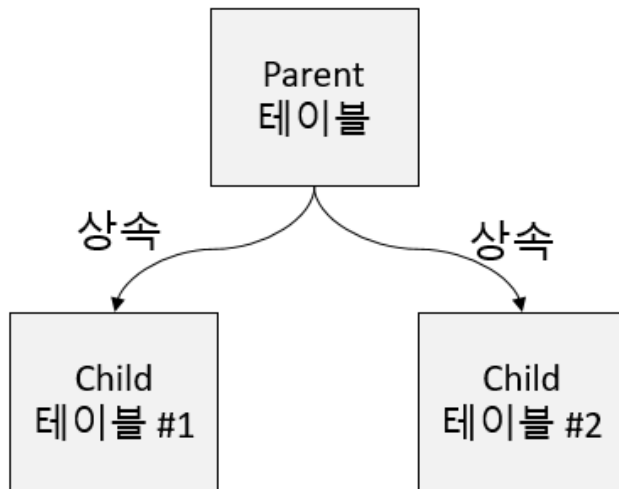
```
select relname, reloptions
from   pg_class
where  relname = 't1';
```

relname	reloptions
t1	{fillfactor=50}

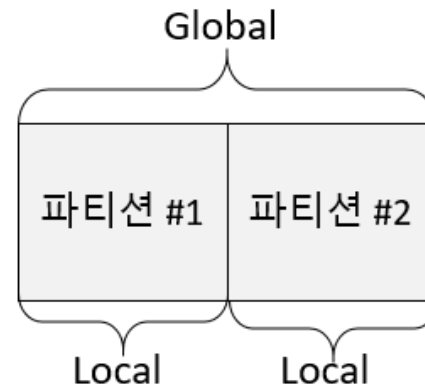
Partition & Tuning

PostgreSQL 파티션의 특징

- 상속 (Inherits)을 이용해서 파티션을 구현함
- 인덱스 정보는 상속되지 않음
- Trigger를 이용해서 파티션 Pruning을 처리함
- 파티션 생성 시에 CHECK 조건을 이용함
- Range, List 파티션을 제공함
- Global Index는 제공하지 않음



(a) PostgreSQL 파티션 개념



(b) ORACLE 파티션 개념

Range 파티션 Pruning 테스트

- 입력 조건에 맞는 파티션만 액세스하는 것을 '파티션 Pruning'이라고 한다.
- 이를 위해서는 `constraint_exclusion` 파라미터를 'on' 또는 'partition'으로 설정해야 한다.
(기본 설정값: partition)

부모 테이블 생성

```
create table range_p1
(
    c1            integer,
    logdate       date,
    dummy         char(100)
);
```

파티션 테이블 생성

```
create table range_p1_y201701
(CHECK( logdate >= DATE '2017-01-01' AND logdate < DATE '2017-02-01' ))
INHERITS (range_p1);

create table range_p1_y201702
(CHECK( logdate >= DATE '2017-02-01' AND logdate < DATE '2017-03-01' ))
INHERITS (range_p1);
```

Range 파티션 Pruning 테스트

트리거 생성

```
CREATE OR REPLACE FUNCTION range_p1_insert_func()
RETURNS TRIGGER AS $$
BEGIN
IF      (NEW.logdate >= DATE '2017-01-01' AND NEW.logdate < DATE '2017-02-01')
THEN INSERT INTO range_p1_y201701 VALUES (NEW.*);
ELSIF (NEW.logdate >= DATE '2017-02-01' AND NEW.logdate < DATE '2017-03-01')
THEN INSERT INTO range_p1_y201702 VALUES (NEW.*);
ELSE RAISE EXCEPTION 'Out of range!';
END IF;
RETURN NULL;
END;
$$
LANGUAGE plpgsql;

-- INSERT 트리거 생성
CREATE TRIGGER range_p1_insert_trig
  BEFORE INSERT ON range_p1
  FOR EACH ROW EXECUTE PROCEDURE range_p1_insert_func();
```

Range 파티션 Pruning 테스트

데이터 입력 및 Analyze

```
do $$
begin
  for i in 1..59 loop
    for j in 1..10000 loop
      insert into range_p1 values
        (j+(10000*(i-1)), to_date('20170101','YYYYMMDD')+i-1, 'dummy');
    end loop;
  end loop;
end$$;
```

```
analyze range_p1_y201701;
analyze range_p1_y201702;
```

```
select relname, relpages, reltuples
from   pg_class where relname like 'range_p1%';
relname      | relpages | reltuples
```

```
-----+-----+-----
range_p1      |         0 |         0
range_p1_y201701 |       5345 |       310010
range_p1_y201702 |       4828 |       279985
```

Range 파티션 Pruning 테스트

데이터 입력 및 Analyze

```
show constraint_exclusion;
constraint_exclusion
-----
partition

explain (costs false)
select sum(c1), min(c1), max(c1)
from   range_p1
where  logdate = to_date('20170212', 'YYYYMMDD');
```

QUERY PLAN

```
Aggregate
-> Append
-> Seq Scan on range_p1
    Filter: (logdate = to_date('20170212'::text, 'YYYYMMDD'::text))
-> Seq Scan on range_p1_y201701
    Filter: (logdate = to_date('20170212'::text, 'YYYYMMDD'::text))
-> Seq Scan on range_p1_y201702
    Filter: (logdate = to_date('20170212'::text, 'YYYYMMDD'::text))
```

파티션 Pruning이
되지 않는다.
왜 일까?

파티션 Pruning은 CHECK 조건과 동일한 형태인 경우만 동작한다.

조건 변경 후의 Explain 결과

```
explain (costs false)
select sum(c1), min(c1), max(c1)
from   range_p1
where  logdate = DATE '20170212';
```

QUERY PLAN

```
-----
Aggregate
-> Append
    -> Seq Scan on range_p1
        Filter: (logdate = '2017-02-12'::date)
    -> Seq Scan on range_p1_y201702
        Filter: (logdate = '2017-02-12'::date)
```

List 파티션 생성 방법

List 파티션 생성

```
create table list_p1
(
    c1          integer,
    code        integer,
    dummy       char(100)
);

create table list_p1_code1 (CHECK( code = 1)) INHERITS (list_p1);
create table list_p1_code2 (CHECK( code = 2)) INHERITS (list_p1);
```

List 파티션 Pruning 테스트

- CHECK 조건을 입력하면 정상적으로 파티션 Pruning이 수행된다.

```
explain select * from list_p1 where code=1;
```

QUERY PLAN

```
-----  
Append  (cost=0.00..8923.00 rows=300001 width=109)  
-> Seq Scan on list_p1  (cost=0.00..0.00 rows=1 width=412)  
    Filter: (code = 1)  
-> Seq Scan on list_p1_code1  (cost=0.00..8923.00 rows=300000 width=109)  
    Filter: (code = 1)
```

서브 파티션을 이용한 튜닝 방안

- 1차 파티션을 통해 IO 범위를 줄인 후에도 여전히 IO 량이 많은 경우에는 2차 파티션을 고려한다.
- PostgreSQL은 상속에 의한 파티션 생성 방식이므로, 이론적으로는 n차 파티션도 가능하다.
- 따라서 업무 속성에 따른 서브 파티션 생성을 통한 IO 튜닝이 가능하다.

부모 테이블 생성

```
create table mp1
(
    c1          integer,
    logdate     date,
    code        integer,
    dummy       char(100)
);
```

서브 파티션을 이용한 튜닝 방안

1차 파티션 테이블 생성 (Range 파티션)

```
create table mp1_y201701
(CHECK ( logdate >= DATE '2017-01-01' AND logdate < DATE '2017-02-01' ))
INHERITS (mp1);
create table mp1_y201702
(CHECK ( logdate >= DATE '2017-02-01' AND logdate < DATE '2017-03-01' ))
INHERITS (mp1);
```

2차 파티션 생성 (List 파티션)

```
-- 서브 파티션 테이블 생성
create table mp1_y201701_code1 (CHECK(code = 1)) INHERITS (mp1_y201701);
create table mp1_y201701_code2 (CHECK(code = 2)) INHERITS (mp1_y201701);

-- 서브 파티션 테이블 생성
create table mp1_y201702_code1 (CHECK(code = 1)) INHERITS (mp1_y201702);
create table mp1_y201702_code2 (CHECK(code = 2)) INHERITS (mp1_y201702);
```

서브 파티션을 이용한 튜닝 방안

트리거 생성

```
CREATE OR REPLACE FUNCTION mp1_insert_func()
RETURNS TRIGGER AS $$
BEGIN
    IF      ( NEW.logdate >= DATE '2017-01-01'  AND
              NEW.logdate <  DATE '2017-02-01') AND ( NEW.code=1 )
        THEN INSERT INTO mp1_y201701_code1 VALUES (NEW.*);
    ELSIF   ( NEW.logdate >= DATE '2017-01-01'  AND
              NEW.logdate <  DATE '2017-02-01') AND ( NEW.code=2 )
        THEN INSERT INTO mp1_y201701_code2 VALUES (NEW.*);
    ELSIF   ( NEW.logdate >= DATE '2017-02-01'  AND
              NEW.logdate <  DATE '2017-03-01') AND ( NEW.code=1 )
        THEN INSERT INTO mp1_y201702_code1 VALUES (NEW.*);
    ELSIF   ( NEW.logdate >= DATE '2017-02-01'  AND
              NEW.logdate <  DATE '2017-03-01') AND ( NEW.code=2 )
        THEN INSERT INTO mp1_y201702_code2 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Out of range!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;

-- INSERT 트리거 생성
CREATE TRIGGER mp1_insert_trig
    BEFORE INSERT ON mp1 FOR EACH ROW EXECUTE PROCEDURE mp1_insert_func();
```

서브 파티션을 이용한 튜닝 방안

데이터 입력

```
do $$
begin
  for i in 1..59 loop
    for j in 1..2 loop
      for k in 1..10000 loop
        insert into mp1
          values (k, to_date('20170101','YYYYMMDD')+i-1, j, 'dummy');
      end loop;
    end loop;
  end loop;
end$$;

analyze mp1_y201701_code1;
analyze mp1_y201701_code2;
analyze mp1_y201702_code1;
analyze mp1_y201702_code2;
```

서브 파티션을 이용한 튜닝 방안

1차 파티션까지 Pruning

```
explain (costs false)
select sum(c1), min(c1), max(c1)
from   mp1
where  logdate between DATE '20170101'
        and          DATE '20170131';
QUERY PLAN
```

Aggregate

-> Append

-> Seq Scan on mp1

Filter: ((logdate >= '2017-01-01'::)
AND (logdate <= '2017-01-31'::date))

-> Seq Scan on mp1_y201701

Filter: ((logdate >= '2017-01-01'::date)
AND (logdate <= '2017-01-31'::date))

-> Seq Scan on mp1_y201701_code1

Filter: ((logdate >= '2017-01-01'::date)
AND (logdate <= '2017-01-31'::date))

-> Seq Scan on mp1_y201701_code2

Filter: ((logdate >= '2017-01-01'::date)
AND (logdate <= '2017-01-31'::date))

서브 파티션을 이용한 튜닝 방안

2차 파티션까지 Pruning

```
explain (costs false)
select sum(c1), min(c1), max(c1)
from   mp1
where  logdate between DATE '20170101'
        and          DATE '20170131'
and    code=2;
```

QUERY PLAN

Aggregate

-> Append

-> Seq Scan on mp1

Filter: ((logdate >= '2017-01-01'::date)
AND (logdate <= '2017-01-31'::date) AND (code = 2))

-> Seq Scan on mp1_y201701

Filter: ((logdate >= '2017-01-01'::date)
AND (logdate <= '2017-01-31'::date) AND (code = 2))

-> **Seq Scan on mp1_y201701_code2**

Filter: ((logdate >= '2017-01-01'::date)
AND (logdate <= '2017-01-31'::date) AND (code = 2))

파티션 Partial 인덱스를 이용한 튜닝 방안

- PostgreSQL은 파티션 Partial 인덱스를 제공한다.
- 즉, 파티션 별로 인덱스 생성 여부를 결정할 수도 있고, 파티션 마다 다른 인덱스 (예, B*Tree, Covering Index, BRIN 등)를 생성할 수도 있다.
- 참고로, 오라클은 12c 버전부터 파티션 Partial 인덱스 기능을 제공한다.

파티션 입력 성능 향상을 위한 Tip

- RULE 보다는 Trigger를 이용한다. 어떤 경우에도 Trigger가 빠르다.
- 자주 입력되는 파티션을 Trigger의 윗 부분에 위치하는 것이 좋다. 그래야 IF THEN ELSE의 부하를 줄일 수 있다.
- 가능하면, 파티션을 지정해서 입력하는 것이 좋다.

PostgreSQL 파티션 Pruning 시의 주의 사항

- 예상과 달리, 파티션 Pruning이 동작하지 않는 경우가 존재한다.
- 이점은 쿼리 작성 시에 매우 주의해야할 점이다.

테스트용 테이블 생성

```
create table log_date_master (id integer, logdate date);

do $$
begin
  for i in 1..365 loop
    insert into log_date_master values (i,to_date('20170101','YYYYMMDD')+i-1);
  end loop;
end$$;
```

PostgreSQL 파티션 Pruning 시의 주의 사항 (IN 절)

- IN 절 내에 Subquery를 사용할 경우에는 파티션 Pruning이 동작하지 않는다.

```
explain (costs false)
select sum(c1), min(c1), max(c1)
from   range_p1
where  logdate in (select logdate from log_date_master where id=1);
```

QUERY PLAN

Aggregate

-> Hash Join

Hash Cond: (range_p1.logdate = log_date_master.logdate)

-> Append

-> Seq Scan on range_p1

-> Seq Scan on range_p1_y201701

-> Seq Scan on range_p1_y201702

-> Hash

-> HashAggregate

Group Key: log_date_master.logdate

-> Seq Scan on log_date_master

Filter: (id = 1)

PostgreSQL 파티션 Pruning 시의 주의 사항 (IN 절)

- IN 절 사용시 파티션 Pruning을 유도하기 위해서는 실제 CHECK 조건을 입력해야 한다.

```
explain (costs false)
select sum(c1), min(c1), max(c1)
from   range_p1
where  logdate in (DATE '2017-01-01');
```

QUERY PLAN

```
-----
Aggregate
  -> Append
        -> Seq Scan on range_p1
            Filter: (logdate = '2017-01-01'::date)
        -> Seq Scan on range_p1_y201701
            Filter: (logdate = '2017-01-01'::date)
```

PostgreSQL 파티션 Pruning 시의 주의 사항 (조인)

- NL 조인으로 수행되면 추가 조건 없이도 파티션 Pruning이 수행된다.
- Hash 조인으로 수행되면 추가 조건을 반드시 입력해야 한다.
- 따라서, 쿼리 작성 후에 Explain 결과 확인 및 추가 조건을 입력할 필요가 있다.

NL 조인의 예

```
explain (costs false)
select sum(b.c1), min(b.c1), max(b.c1)
from   log_date_master a, range_p1 b
where  a.logdate = DATE '2017-01-15'
and    a.logdate = b.logdate;
```

QUERY PLAN

Aggregate

-> **Nested Loop**

-> Seq Scan on log_date_master a

Filter: (logdate = '2017-01-15'::date)

-> Materialize

-> Append

-> Seq Scan on range_p1 b

Filter: (logdate = '2017-01-15'::date)

-> **Seq Scan on range_p1_y201701 b_1**

Filter: (logdate = '2017-01-15'::date)

NL 조인은 상수 값이 전달되므로 파티션 Pruning이 수행된다.

PostgreSQL 파티션 Pruning 시의 주의 사항 (조인)

Hash 조인의 예

```
explain (costs false)
select sum(b.c1), min(b.c1), max(b.c1)
from   log_date_master a, range_p1 b
where  a.logdate BETWEEN DATE '2017-01-15' AND DATE '2017-01-17'
and    a.logdate = b.logdate;
```

QUERY PLAN

Aggregate

-> **Hash Join**

Hash Cond: (b.logdate = a.logdate)

-> Append

-> Seq Scan on range_p1 b

-> **Seq Scan on range_p1_y201701 b_1**

-> **Seq Scan on range_p1_y201702 b_2**

-> Hash

-> Seq Scan on log_date_master a

Filter: ((logdate >= '2017-01-15'::date) AND
(logdate <= '2017-01-17'::date))

Hash 조인이 수행됨에
따라, Partition
Pruning이 동작하지
않는다.

PostgreSQL 파티션 Pruning 시의 주의 사항 (조인)

- Hash 조인으로 수행될 때를 고려해서, 상수 조건을 추가하는 것을 습관화 하는 것이 좋다.

```
explain (costs false)
select sum(b.c1), min(b.c1), max(b.c1)
from   log_date_master a, range_p1 b
where  a.logdate BETWEEN DATE '2017-01-15' AND DATE '2017-01-17'
and    a.logdate = b.logdate
and    b.logdate BETWEEN DATE '2017-01-15' AND DATE '2017-01-17';
      QUERY PLAN
```

Aggregate

-> **Hash Join**

Hash Cond: (a.logdate = b.logdate)

-> Seq Scan on log_date_master a

Filter: ((logdate >= '2017-01-15'::date)
AND (logdate <= '2017-01-17'::date))

-> Hash

-> Append

-> Seq Scan on range_p1 b

Filter: ((logdate >= '2017-01-15'::date) AND
(logdate <= '2017-01-17'::date))

-> **Seq Scan on range_p1_y201701 b_1**

Filter: ((logdate >= '2017-01-15'::date) AND
(logdate <= '2017-01-17'::date))

상수 조건 추가 후에
Partition Pruning이
동작하는 것을 알 수
있다.

PostgreSQL 10 - 파티션 관련 New Feature

- PostgreSQL 10부터 오라클과 유사한 파티션 생성 문법을 제공한다.
- 이때, 트리거는 생성할 필요가 없다. (이점은 관리측면의 편의성을 제공한다)
- 단, 파티션 Pruning은 기존과 같은 한계가 존재한다. (즉, 해시 조인 시에는 상수 조건 추가 필요)

PostgreSQL 10 - 파티션 관련 New Feature

■ 파티션 생성 방법

```
create table p1 (  
    c1            integer,  
    logdate       date,  
    dummy         char(10)  
) partition by range (logdate);
```

Partition by 문법을
지원한다.

```
create table p2 (  
    c1            integer,  
    logdate       date,  
    amount        integer,  
    dummy         char(10)  
) partition by range (logdate);
```

Partition of를 이용해서
개별 파티션을
생성한다.

Range 파티션의 'To'는
LESS THAN을 의미한다.
즉, 2017-02-01보다 작은
일자를 의미한다.

```
create table p1_y201701 partition of p1 for values from ('2017-01-01') to ('2017-02-01');  
create table p1_y201702 partition of p1 for values from ('2017-02-01') to ('2017-03-01');  
create table p1_y201703 partition of p1 for values from ('2017-03-01') to ('2017-04-01');  
create table p1_y201704 partition of p1 for values from ('2017-04-01') to ('2017-05-01');  
  
create table p2_y201701 partition of p2 for values from ('2017-01-01') to ('2017-02-01');  
create table p2_y201702 partition of p2 for values from ('2017-02-01') to ('2017-03-01');  
create table p2_y201703 partition of p2 for values from ('2017-03-01') to ('2017-04-01');  
create table p2_y201704 partition of p2 for values from ('2017-04-01') to ('2017-05-01');
```

PostgreSQL 10 - 파티션 관련 New Feature

■ 파티션 Pruning 테스트

```
explain (costs false)
select count(*)
from   p1 a, p2 b
where  a.logdate = b.logdate
and    a.c1      = b.c1
and    a.logdate between DATE '2017-01-15'
                        and DATE '2017-01-31';
      QUERY PLAN
```

Aggregate

-> **Hash Join**

Hash Cond: ((b.logdate = a.logdate) AND (b.c1 = a.c1))

-> Append

-> Seq Scan on **p2_y201701 b**

-> Seq Scan on **p2_y201702 b_1**

-> Seq Scan on **p2_y201703 b_2**

-> Seq Scan on **p2_y201704 b_3**

-> Hash

-> Append

-> Seq Scan on p1_y201701 a

Filter: ((logdate >= '2017-01-15'::date) AND
(logdate <= '2017-01-31'::date))

Hash 조인 시에는 이전 버전과 같은 문제점이 있다.

파티션 성능 테스트 (INSERT - Case #1)

- 9.6 까지는 파티션이 1개인 경우에도 일반 테이블보다 입력 속도가 3배 이상 느리다. 그 이유는 매 건마다 Trigger 함수를 호출하는 부하 때문이다.
- CPU 성능에 좋을수록 이 차이는 감소한다.
- 파티션 개수가 많아질 수록 입력 성능은 저하된다.
- 이 문제는 PG10 버전에서도 여전히 발생한다.

구분	100만건 입력 속도 (초)		1000만건 입력 속도 (초)	
	PG 9.6	PG 10	PG 9.6	PG 10
일반테이블	7	7	68	74
파티션 테이블 1개	25	8	255	85
파티션 테이블 10개	28	18	271	186
파티션 테이블 100개	149	138	1,480	1,395
파티션 테이블 200개	483	290		

1

2

파티션 성능 테스트 (INSERT - Case #2)

- 상속을 이용한 파티션은 Trigger 조정을 통해서 입력 성능을 향상 시킬 수 있다.
- 하지만, Native 파티션 (PG 10 버전)은 입력 파티션 위치에 따른 성능 차이가 거의 없다.

구분	100만건 입력 속도 (초)	
	PG 9.6	PG10
파티션 테이블 200개 균등 입력	483	290
파티션 테이블 200개 중 첫 번째 파티션에만 입력	32	284
파티션 테이블 200개 중 마지막 파티션에만 입력	941	280

토의!

Part-II.

PostgreSQL

쿼리 옵티마이저

CBO 개요

Optimizer란?

- Optimizer란?

- ✓ 쿼리 성능을 최적화하기 위한 프로그램 (또는 로직)

- Optimizer의 종류

Rule-Based
Optimizer

Cost-Based
Optimizer

PostgreSQL은
CBO를 사용

COST란?

- ✓ 파라미터와 로직을 이용해서 계산된 숫자
- ✓ COST 값으로 IO 블록 수와 CPU 사용 시간을 계산할 수는 없음
- ✓ 단지, COST가 낮을수록 효율적이고, 높을수록 비효율적이라고 추정함

COST 계산에 이용되는 파라미터

구분	파라미터	기본 설정값
IO 비용	seq_page_cost	1
	random_page_cost	4
CPU 비용	cpu_tuple_cost	0.01
	cpu_index_tuple_cost	0.005
	cpu_operator_cost	0.0025

IO 비용 계산을 위한 파라미터

- **seq_page_cost**

- ✓ Seq Scan 방식으로 1 블록을 읽는 비용

- **random_page_cost**

- ✓ Index Scan 방식으로 1 블록을 읽는 비용
- ✓ 인덱스 Root 블록과 Branch 블록을 제외

CPU 비용 계산을 위한 파라미터

- **cpu_tuple_cost**
 - ✓ Seq Scan 수행 시에 1개 레코드를 액세스하는 비용
- **cpu_index_tuple_cost**
 - ✓ Index Scan 수행 시에 1개 레코드를 액세스하는 비용
- **cpu_operator_cost**
 - ✓ 레코드 1개를 필터 처리하는 비용

Seq Scan 비용 계산

- ✓ 매우 단순한 방식으로 계산함

Seq Scan 비용 계산 예제 (1)

```
select relpages, reltuples from pg_class where relname='t1';
relpages | reltuples
```

```
-----+-----
      443 |    100000
```

```
explain select * from t1;
```

```
          QUERY PLAN
```

```
-----
Seq Scan on t1  (cost=0.00..1443.00 rows=100000 width=8)
```

COST=

```
select relpages * current_setting('seq_page_cost')::float +
       reltuples * current_setting('cpu_tuple_cost')::float
from   pg_class where relname='t1';
```

= 443 * 1.0 + 100,000 * 0.01

= 1,443

Seq Scan 비용 계산 예제 (2)

```
explain select * from t1 where c1 <= 300;  
               QUERY PLAN
```

```
-----  
Seq Scan on t1  (cost=0.00..1693.00 rows=277 width=8)  
  Filter: (c1 <= 300)
```

COST=

```
select relpages * current_setting('seq_page_cost')::float +  
       reltuples * current_setting('cpu_tuple_cost')::float +  
       reltuples * current_setting('cpu_operator_cost')::float  
from   pg_class where relname='t1';
```

= 443 * 1.0 + 100,000 * 0.01 + 100,000 * 0.0025

= 1,693

Statistics

통계 정보 생성 단위

- ✓ 데이터베이스, 테이블, 칼럼 레벨로 통계 정보 생성 가능
- ✓ 스키마, 인덱스 레벨은 지원하지 않음

- 데이터베이스 레벨

```
analyze;
```

- 테이블 레벨

```
analyze {테이블명};
```

- 칼럼 레벨

```
analyze {테이블명} {(칼럼명) ..};
```

- PG_CLASS

칼럼 명	설명
relpages	블록 수
reltuples	레코드 수

주요 통계 정보 확인

■ PG_STATS

칼럼 명	설명
null_frac	NULL 값의 비율을 의미한다.
avg_width	칼럼 평균 길이를 의미한다.
n_distinct	NDV (Number of Distinct Value)를 의미하며, NDV 값에 따라서 양수 또는 음수 값을 갖는다. 만일 NDV가 테이블 건수 대비 10% 이내이면 NDV 그대로 표시한다. 하지만 NDV가 10% 이상이면 $NDV = -(NDV / \text{레코드 수})$ 공식을 적용한다.
correlation	칼럼 정렬 상태를 나타낸다. -1~1 사이의 값으로 표시되며 완벽히 정렬된 상태면 1, 완벽히 역순으로 정렬된 상태면 -1 이다.

Autovacuum이 자동으로 통계 정보를 갱신하는 기준

- 아래의 파라미터를 이용해서 통계 정보를 갱신한다.

파라미터 명	설명	설정값
autovacuum	autovacuum 프로세스 사용 여부	on
autovacuum_analyze_scale_factor	테이블 내의 레코드 변경 비율	0.1
autovacuum_analyze_threshold	최소 변경 레코드 수	50

- 테이블 별 설정도 가능하다.

```
alter table t1 set (autovacuum_analyze_scale_factor = 0.0);  
alter table t1 set (autovacuum_analyze_threshold = 100000);
```

```
alter table t1 set (autovacuum_analyze_scale_factor = 0.1);  
alter table t1 set (autovacuum_analyze_threshold = 0);
```

Explain

Explain 사용 모드

- Explain은 크게 2가지 모드로 사용할 수 있다.
 - 1) 예측 모드: 실제 수행은 하지 않고 예상 실행 계획을 제공한다.
 - 2) 실행 모드: 실제 수행을 한 후에 실행 계획, 수행 시간, IO 블록 수를 제공한다.

예측 모드

■ 예측 모드 사용법

- 쿼리 앞에 explain 키워드만 추가하면 된다.
- 통계 정보를 참고해서 COST, 예상 ROWS, 칼럼 길이 정보를 제공한다.

```
explain select * from t2;
```

```
QUERY PLAN
```

```
-----
```

```
Seq Scan on t2  (cost=0.00..18334.00 rows=1000000 width=37)
```

■ 실행 모드 사용법

- 쿼리 앞에 explain analyze 키워드만 입력하면 된다.
- IO 블록 수를 확인하기 위해서는 buffers 키워드를 추가한다.
- Step 별 실제 수행 시간, 실제 ROWS, Loop 횟수, Planning Time, Execution Time을 제공한다.
- 총 수행 시간은 Planning Time + Execution Time이다.
- Execution Time에 화면 Display 시간은 포함되지 않는다.
- 실제로 쿼리를 수행하므로 DML 수행 시 유의해야 한다.

```
explain (analyze, buffers) select * from t2;
                        QUERY PLAN
-----
Seq Scan on t2   (cost=0.00..18334.00 rows=1000000 width=37)
    (actual time=0.008..173.641 rows=1000000 loops=1)
    Buffers: shared hit=8334
Planning time: 0.023 ms
Execution time: 286.944 ms
```

Explain 결과 중 주요 항목 설명

```
explain (analyze, buffers) select * from t2 where c1=1;  
QUERY PLAN
```

```
-----  
Seq Scan on t2 (cost=0.00..20834.00 rows=98 width=37)  
  (actual time=1.087..101.167 rows=100 loops=1)  
    Filter: (c1 = 1)  
    Rows Removed by Filter: 999900  
    Buffers: shared hit=8334
```

Rows Removed by Filter
수치가 크면 인덱스
생성을 고려해야 한다.

▪ Startup Cost 및 Total Cost

- Startup Cost는 첫 번째 레코드를 fetch 하는데 드는 비용
- Total Cost는 전체 레코드를 fetch 하는데 드는 비용

▪ Actual Time

- 첫 번째 레코드를 fetch 하는데 소요된 시간과 전체 레코드를 fetch 하는데 소요된 시간을 제공한다. 단위는 Milli-second (1/1,000초)이다.

▪ rows


- 예상 로우 수와 실제 로우 수를 제공한다.

▪ Buffers

- IO 횟수를 의미한다. **shared hit**: 메모리 IO 횟수, **shared read**: 디스크 IO 횟수

Explain 결과 읽는 방법

- 원칙 1: 안쪽부터 읽는다.
- 원칙 2: 조인 시에는 OUTER 테이블이 위에 위치한다.



원칙 1은 ORACLE과 동일한다. 하지만 원칙 2는 ORACLE과 일부 다르다.

이 부분이 ORACLE에 익숙한 사용자에게는 다소 헷갈린 부분이다.

원칙 1: 안쪽부터 읽는다.

- 이 원칙은 매우 쉽고 명확하다.

```
drop table t1;  
create table t1 (c1 integer, dummy char(1000));
```

```
insert into t1 select mod(i,1000)+1, 'dummy'  
from   generate_series(1,10000) a(i);
```

```
create index t1_idx01 on t1(c1);  
analyze t1;
```

```
explain select * from t1 where c1 between 1 and 10;  
               QUERY PLAN
```

```
-----  
(1)  Bitmap Heap Scan on t1   (cost=5.35..330.36 rows=104 width=1008)  
      Recheck Cond: ((c1 >= 1) AND (c1 <= 10))  
(2)   -> Bitmap Index Scan on t1_idx01 (cost=0.00..5.33 rows=104 width=0)  
        Index Cond: ((c1 >= 1) AND (c1 <= 10))
```

수행 순서는?

원칙 2: 조인 시에는 OUTER 테이블이 위에 위치한다.

- 이 원칙은 ORACLE에 비해서는 조금 헛갈리다.
- NL 조인은 위에서 부터 읽는다. (ORACLE과 같다)
- 해시 조인은 아래에서부터 읽는다. (ORACLE과 반대다)

조인 방법	OUTER 테이블	Inner 테이블	먼저 액세스되는 테이블
NL 조인	Driving 테이블	Inner 테이블	Driving 테이블
해시 조인	Probe 테이블	Build 테이블	Build 테이블

PostgreSQL는
OUTER 테이블이
위에 위치한다.

ORACLE은 먼저
액세스되는
테이블이 위에
위치한다.

테스트 데이터 생성

```
drop table t1;
drop table t2;
drop table t3;

create table t1 (c1 integer, dummy char(1000));
create table t2 (c1 integer, dummy char(1000));
create table t3 (c1 integer, dummy char(1000));

insert into t1 select generate_series(1,10), 'dummy';
insert into t2 select generate_series(1,1000), 'dummy';
insert into t3 select generate_series(1,10000), 'dummy';

create index t2_idx01 on t2(c1);
create index t3_idx01 on t3(c1);

analyze t1;
analyze t2;
analyze t3;
```

테이블 크기 순서는
T3 > T2 > T1

NL 조인 Explain 예제

```
set enable_mergejoin=off;  
set enable_hashjoin=off;
```

```
explain (costs false)
```

```
select *  
from   t1 a, t2 b, t3 c  
where  a.c1 = b.c1  
and    b.c1 = c.c1;
```

```
QUERY PLAN
```

Nested Loop

Join Filter: (a.c1 = b.c1)

(1) -> **Nested Loop**

(2) -> Seq Scan on t1 a

(3) -> Index Scan using t3_idx01 on t3 c
Index Cond: (c1 = a.c1)

(4) -> Index Scan using t2_idx01 on t2 b
Index Cond: (c1 = c.c1)

수행 순서는?

해시 조인 Explain 예제 (#1)

```
set enable_hashjoin=on;  
set enable_nestloop=off;
```

```
explain (costs false)  
select *  
from   t1 a, t2 b  
where  a.c1 = b.c1;  
          QUERY PLAN
```

```
-----  
Hash Join  
  Hash Cond: (b.c1 = a.c1)  
(1)  ->  Seq Scan on t2 b  
(2)  ->  Hash  
(3)      ->  Seq Scan on t1 a
```

수행 순서는?

해시 조인 Explain 예제 (#2)

```
explain (costs false)
select *
from   t1 a, t2 b, t3 c
where  a.c1 = b.c1
and    b.c1 = c.c1;

                        QUERY PLAN
```

Hash Join

Hash Cond: (c.c1 = a.c1)

- (1) -> Seq Scan on t3 c
- (2) -> Hash
- (3) -> **Hash Join**
 - Hash Cond: (b.c1 = a.c1)
- (4) -> Seq Scan on t2 b
- (5) -> Hash
- (6) -> Seq Scan on t1 a

수행 순서는?

Access method

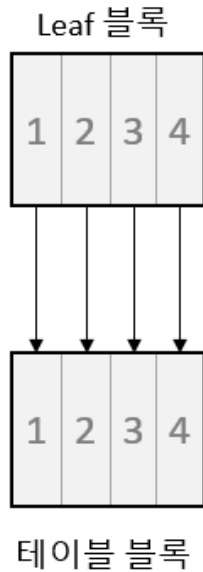
Seq Scan 방식

- Seq Scan은 테이블을 Full Scan 하면서 레코드를 읽는 방식이다.
- 인덱스가 존재하지 않거나, 인덱스가 존재하더라도 읽어야 할 범위가 넓은 경우에 선택한다.

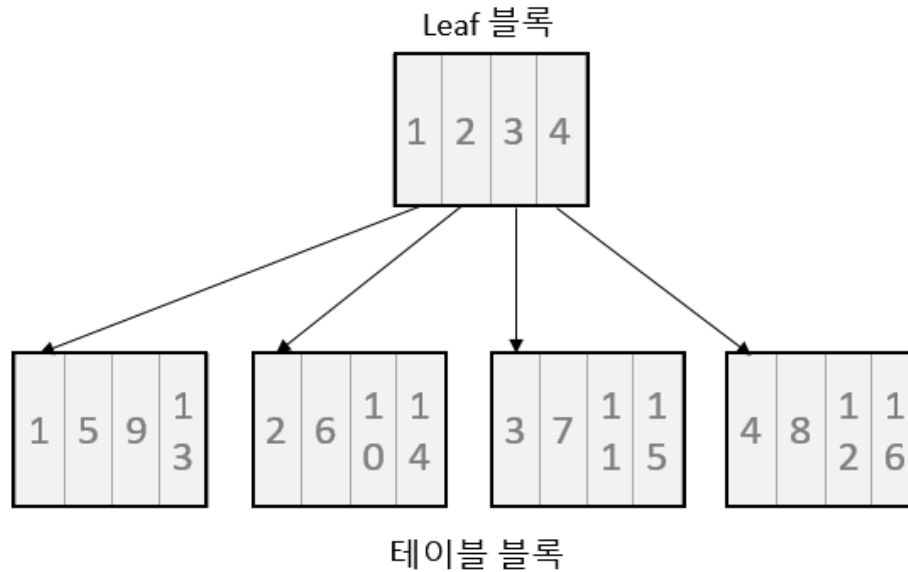
Index Scan 방식의 특징

- Index Scan은 인덱스 Leaf 블록에 저장된 키를 이용해서 테이블 레코드를 액세스하는 방식이다.
- 인덱스 키 순서대로 출력된다.
- 레코드 정렬 상태에 따라서 테이블 블록 액세스 횟수가 크게 차이 난다.

인덱스와 테이블 정렬 순서가
가장 잘 일치하는 경우



인덱스와 테이블 정렬 순서가
가장 불일치하는 경우



Index Scan 방식의 Explain 예제

- Index Scan은 인덱스 Leaf 블록에 저장된 키를 이용해서 테이블 레코드를 액세스하는 방식이다.
- 인덱스 키 순서대로 출력된다.
- 레코드 정렬 상태에 따라서 테이블 블록 액세스 횟수가 크게 차이 난다.

```
drop table t1;
create table t1 (c1 integer, dummy char(100));
insert into t1 select generate_series(1,58000), 'dummy';
create index t1_idx01 on t1(c1);
analyze t1;
```

```
explain (costs false, analyze, buffers)
select * from t1 where c1 between 1 and 4000;
                                QUERY PLAN
```

```
Index Scan using t1_idx01 on t1 (actual time=0.012..1.567 rows=4000 loops=1)
  Index Cond: ((c1 >= 1) AND (c1 <= 4000))
  Buffers: shared hit=71 read=10
Planning time: 0.191 ms
Execution time: 2.042 ms
```

Index Scan 방식의 Explain
결과는 테이블과 인덱스를
구분해서 제공하지는 않음

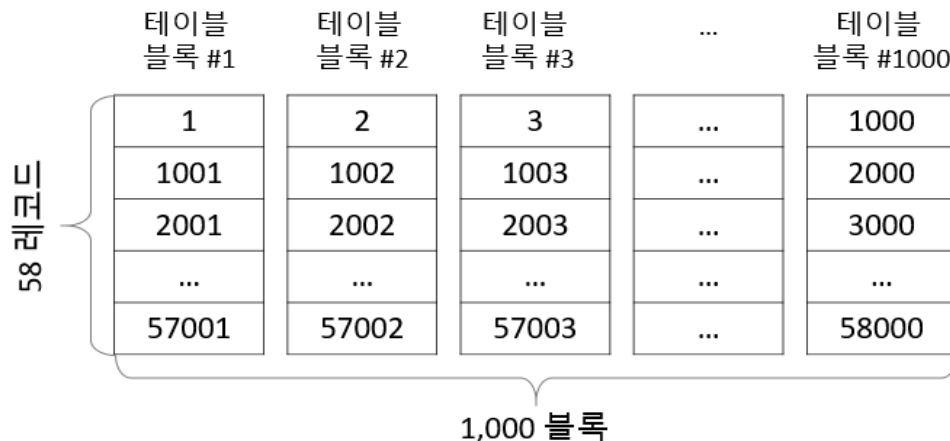
Bitmap Index Scan 방식의 특징

- 테이블 랜덤 액세스 횟수를 줄이기 위해 고안된 방식이다.
- Index Scan 방식과 Bitmap Index Scan 방식을 결정하는 기준은 인덱스 칼럼의 Correlation 값이다.
- Correlation이란 인덱스 칼럼에 대한 테이블 레코드의 정렬 상태이다.
- 즉, Correlation이 좋으면 Index Scan 방식을, 나쁘면 Bitmap Index Scan 방식을 사용한다.
- Bitmap Index Scan 방식은 액세스할 블록들을 블록 번호 순으로 정렬한 후에 액세스한다.
- 이로 인해, 테이블 랜덤 액세스 횟수가 크게 줄어든다. (블록당 1회)
- 테이블 블록 번호 순으로 액세스하므로, 인덱스 키 순서대로 출력되지 않는다.

테스트 환경 구성

```
drop table t2;
create table t2 (c1 integer, dummy char(100));
-- 테이블에 1,1001,2001...2,1002,2002...순으로 입력되도록 한다.
do $$
begin
for i in 1..1000 loop
    for j in 0..57 loop
        insert into t2 values (i+(j*1000),'dummy');
    end loop;
end loop;
end$$;

create index t2_idx01 on t2(c1);
analyze t2;
```



Bitmap Index Scan 방식의 Explain 예제

- C1 조건이 1~1000 이면 테이블 블록을 1000 번 액세스한다.

```
explain (costs false, analyze, buffers)
select * from t2 where c1 between 1 and 1000;
                        QUERY PLAN
-----
Bitmap Heap Scan on t2 (actual time=0.264..0.979 rows=1000 loops=1)
  Recheck Cond: ((c1 >= 1) AND (c1 <= 1000))
  Heap Blocks: exact=1000
  Buffers: shared hit=1002 read=2
-> Bitmap Index Scan on t2_idx01
   Index Cond: ((c1 >= 1) AND (c1 <= 1000))
   Buffers: shared hit=2 read=2
```

- C1 조건이 1~4000 이면 테이블 블록을 몇 회 액세스할까?

```
explain (costs false, analyze, buffers)
select * from t2 where c1 between 1 and 4000;
```

- Index Scan 방식으로 수행하면 테이블 블록을 몇 회 액세스할까?

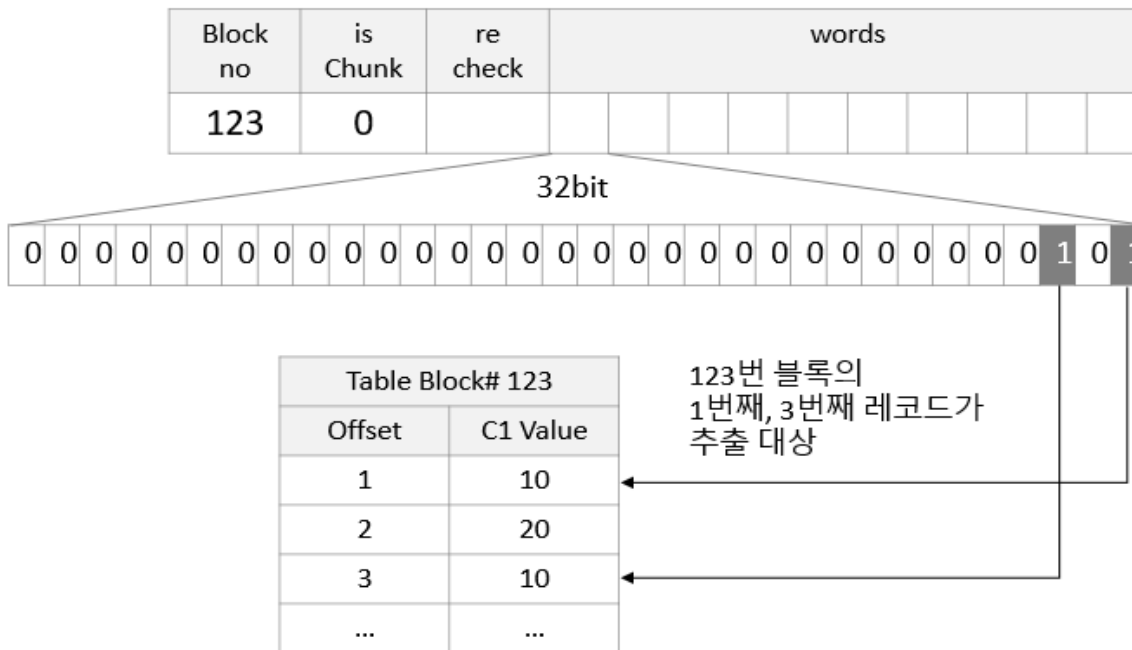
```
set enable_bitmapscan=off;
set enable_seqscan=off;

explain (costs false, analyze, buffers)
select * from t2 where c1 between 1 and 4000;
```

Lossy 모드의 이해

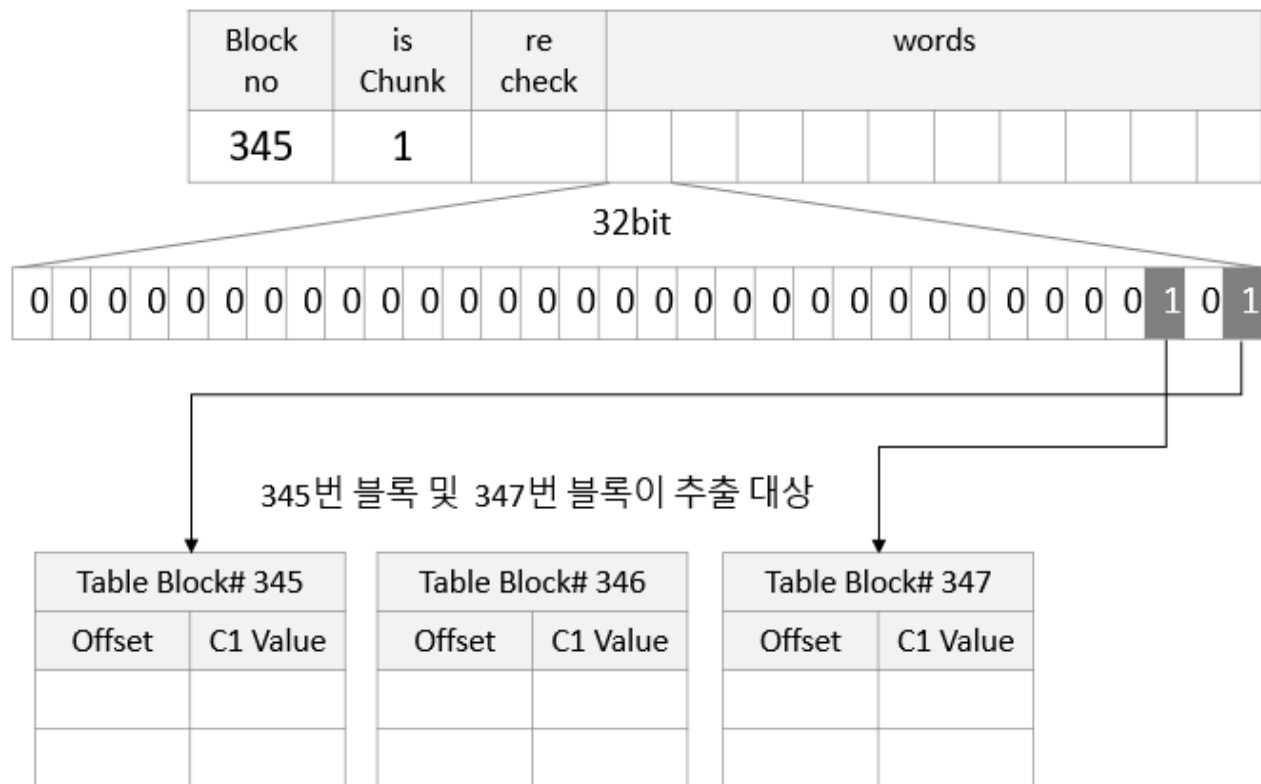
- Bitmap Index Scan 방식은 비트맵을 이용해서 처리된다.
- 이때, 비트맵 정보는 Backend 프로세스 메모리 내에 저장된다.
- 만일 메모리 공간이 부족하면 exact 모드에서 lossy 모드로 전환한다.
- lossy 모드는 exact 모드에 비해서 느리다.

exact 모드는 비트맵 내의 1개의 비트가 1개의 레코드를 가리킨다.



Lossy 모드의 이해

lossy 모드는 비트맵 내의 1개의 비트가 1개의 블록을 가리킨다.



Lossy 모드의 이해

테스트 환경 구성

```
drop table t3;
create table t3 (c1 integer, dummy char(100));

-- 100,000 블록에 58건씩 총 580만 건을 입력한다.
do $$
begin
  for i in 1..100000 loop
    for j in 0..57 loop
      insert into t3 values (i+(j*100000), 'dummy');
    end loop;
  end loop;
end$$;
```

퀴즈 - Lossy가 나타나는 Explain 결과를 해석해보자.

```
explain (costs false, analyze, buffers)
select * from t3 where c1 between 1 and 1000000;
               QUERY PLAN
-----
Bitmap Heap Scan on t3 (actual time=233.466..1012.086 rows=1000000 loops=1)
  Recheck Cond: ((c1 >= 1) AND (c1 <= 1000000))
  Rows Removed by Index Recheck: 2531424
  Heap Blocks: exact=47262 lossy=52738
  Buffers: shared hit=41 read=102694
    -> Bitmap Index Scan on t3_idx01
        (actual time=220.747..220.747 rows=1000000 loops=1)
        Index Cond: ((c1 >= 1) AND (c1 <= 1000000))
        Buffers: shared hit=40 read=2695
Planning time: 0.438 ms
Execution time: 1111.011 ms
```

- 1) 테이블 블록 IO 총횟수는? 100,000
- 2) Rows Removed by Index Recheck 수치인 2,531,424는 어떻게 계산됐을까? $52,738 * 48$

CLUSTER 명령어를 이용한 테이블 재구성

- 특정 인덱스 칼럼 기준으로 테이블을 재 정렬해서 다시 생성하고 싶다면 CLUSTER 명령어를 사용하면 된다.
- 다만, 이때도 Vacuum FULL과 동일하게 SELECT와도 락이 호환되지 않는다는 점을 유의해야 한다.

```
CLUSTER T3 USING T3_IDX01;  
analyze t3;
```


Index Only Scan 방식의 특징

- Covering Index를 이용하는 것을 Index Only Scan 방식이라고 한다.
- Covering Index란 SELECT 칼럼 및 WHERE 조건을 모두 포함하는 인덱스를 의미한다.
- Covering Index의 장점은 테이블 랜덤 액세스를 제거할 수 있다는 것이다.
- 단, Vacuum을 수행해야만 Index Only Scan 방식으로 동작한다.

```
SELECT C2, C3
FROM   T1
WHERE  C1 BETWEEN :b1 AND :b2;

CREATE T1_IDX01 ON T1 (C1, C2, C3);
```

쿼리 수행 시에 필요한 모든
칼럼을 결합한 인덱스를
Covering 인덱스라고 한다.

테스트 환경 구성

```
drop table t3;

create table t3 (c1 integer, c2 integer, c3 integer, dummy char(100));

insert into t3 select i, i, i, 'dummy'
from generate_series(1,100000) a(i);

create index t3_idx01 on t3(c1, c2, c3);

analyze t3;
```

Vacuum 수행 전의 Explain 결과 확인

- C1, C2, C3 칼럼으로 구성된 인덱스를 생성했지만 Index Only Scan 방식으로 수행되지 않았다.
- 이는 Vacuum을 수행하지 않았기 때문이다.

```
explain (costs false, analyze, buffers)
select c2, c3 from t3 where c1 between 1 and 10000;
                                QUERY PLAN
-----
Bitmap Heap Scan on t3 (actual time=0.677..2.528 rows=10000 loops=1)
  Recheck Cond: ((c1 >= 1) AND (c1 <= 10000))
  Heap Blocks: exact=182
  Buffers: shared hit=223
    -> Bitmap Index Scan on t3_idx01 (actual time=0.656..0.656 rows=10000..)
      Index Cond: ((c1 >= 1) AND (c1 <= 10000))
      Buffers: shared hit=41
Planning time: 0.066 ms
Execution time: 3.356 ms
```

Vacuum 수행 후의 Explain 결과 확인

- Vacuum 후에는 Index Only Scan 방식으로 동작한다.
- 내부적으로 보면, Visibility Map의 ALL_VISIBLE 비트가 1이면 Index Only Scan 방식으로 동작한다.

```
Vacuum t3;
```

```
explain (costs false, analyze, buffers)
select c2, c3 from t3 where c1 between 1 and 10000;
```

```
QUERY PLAN
```

```
-----
Index Only Scan using t3_idx01 on t3 (actual time=0.012..1.796 rows=10000..)
  Index Cond: ((c1 >= 1) AND (c1 <= 10000))
  Heap Fetches: 0
  Buffers: shared hit=42
Planning time: 0.084 ms
Execution time: 2.678 ms
```

액세스 방식을 제어하는 방법

- SET 절을 이용해서 액세스 방식을 제어할 수 있다.

항목	설명
enable_seqscan	Seq Scan 사용 여부를 제어한다.
enable_indexscan	Index Scan 사용 여부를 제어한다.
enable_bitmapscan	Bitmap Index Scan 사용 여부를 제어한다.
enable_indexonlyscan	Index Only Scan 사용 여부를 제어한다.

Join method

PostgreSQL에서 지원하는 조인 방법

- Nested Loop 조인
- Sort Merge 조인
- 해시 조인 (Hybrid 해시 조인 지원)

어떤 조인 방법을 선택하는 것이 좋을까?

- 조인 방법을 선택할 때는 조인의 특성을 이해하는 것이 필요하다.

조인 방법	수행 방식	튜닝 포인트
NL 조인	<ul style="list-style-type: none">Driving 테이블을 액세스한 후리턴된 결과를 반복적으로 Inner 테이블을 액세스한다.	<ul style="list-style-type: none">Driving 테이블 액세스 범위 및 방법Inner 테이블 액세스 효율Random Access에 취약
해시 조인	<ul style="list-style-type: none">Hash Build 테이블에 대한 해시 메모리 작업을 수행한다.Probe 테이블을 액세스하면서 조인 작업을 수행한다.	<ul style="list-style-type: none">Hash Build 테이블의 크기Hash 충돌
Sort Merge 조인	<ul style="list-style-type: none">첫 번째 테이블을 Sorting 한다. (인덱스가 없는 경우)두 번째 테이블을 Sorting 한 후, 집합 간의 결과를 비교한다.	<ul style="list-style-type: none">Sorting 부하 (인덱스 없는 경우)Random IO 부하 (인덱스 있는 경우)

NL 조인 시에 Materialize가 발생하면 인덱스 생성을 고려해야 한다.

- NL 조인 시에 연결 고리에 인덱스가 없으면 Materialize 오퍼레이션이 발생할 수 있다.
- 이는 보완책이지 해결책이 아니다. 따라서 인덱스 생성을 고려해야 한다.

테스트 환경 구성

```
drop table t1;
drop table t2;

create table t1 (c1 integer, c2 integer, dummy char(100));
create table t2 (c1 integer, c2 integer, dummy char(100));

insert into t1 select i, i, 'dummy' from generate_series(1,1000000) a(i);
insert into t2 select i, i, 'dummy' from generate_series(1,100000) a(i);

create index t1_idx01 on t1(c1);
```

NL 조인 시 Materialize 발생 예제

```
set enable_hashjoin=off;  
set enable_mergejoin=off;
```

```
explain (costs false, timing false, analyze, buffers)  
select *  
from   t1 a, t2 b  
where  a.c1 between 1 and 10  
and   a.c2 = b.c2;
```

조인 조건에
인덱스가 없음

QUERY PLAN

```
-----  
Nested Loop (actual rows=10 loops=1)  
  Join Filter: (a.c2 = b.c2)  
  Rows Removed by Join Filter: 999990  
  Buffers: shared hit=1726 read=3  
-> Seq Scan on t2 b (actual rows=100000 loops=1)  
    Buffers: shared hit=1725  
-> Materialize (actual rows=10 loops=100000)  
    Buffers: shared hit=1 read=3  
      -> Bitmap Heap Scan on t1 a (actual rows=10 loops=1)  
        Recheck Cond: ((c1 >= 1) AND (c1 <= 10))  
        Heap Blocks: exact=1  
        Buffers: shared hit=1 read=3  
          -> Bitmap Index Scan on t1_idx01 (actual rows=10 loops=1)  
            Index Cond: ((c1 >= 1) AND (c1 <= 10))
```

Materialize 발생

Planning time: 0.224 ms

Execution time: 117.460 ms

인덱스 생성 후 Explain 결과 확인

```
create index t2_idx02 on t2(c2);
```

```
explain (costs false, timing false, analyze, buffers)
select *
from   t1 a, t2 b
where  a.c1 between 1 and 10
and    a.c2 = b.c2;
```

QUERY PLAN

Nested Loop (actual rows=10 loops=1)

Buffers: shared hit=32 read=2

-> **Index Scan using t1_idx01 on t1 a** (actual rows=10 loops=1)

Index Cond: ((c1 >= 1) AND (c1 <= 10))

Buffers: shared hit=4

-> **Index Scan using t2_idx02 on t2 b** (actual rows=1 loops=10)

Index Cond: (c2 = a.c2)

Buffers: shared hit=28 read=2

Planning time: 0.363 ms

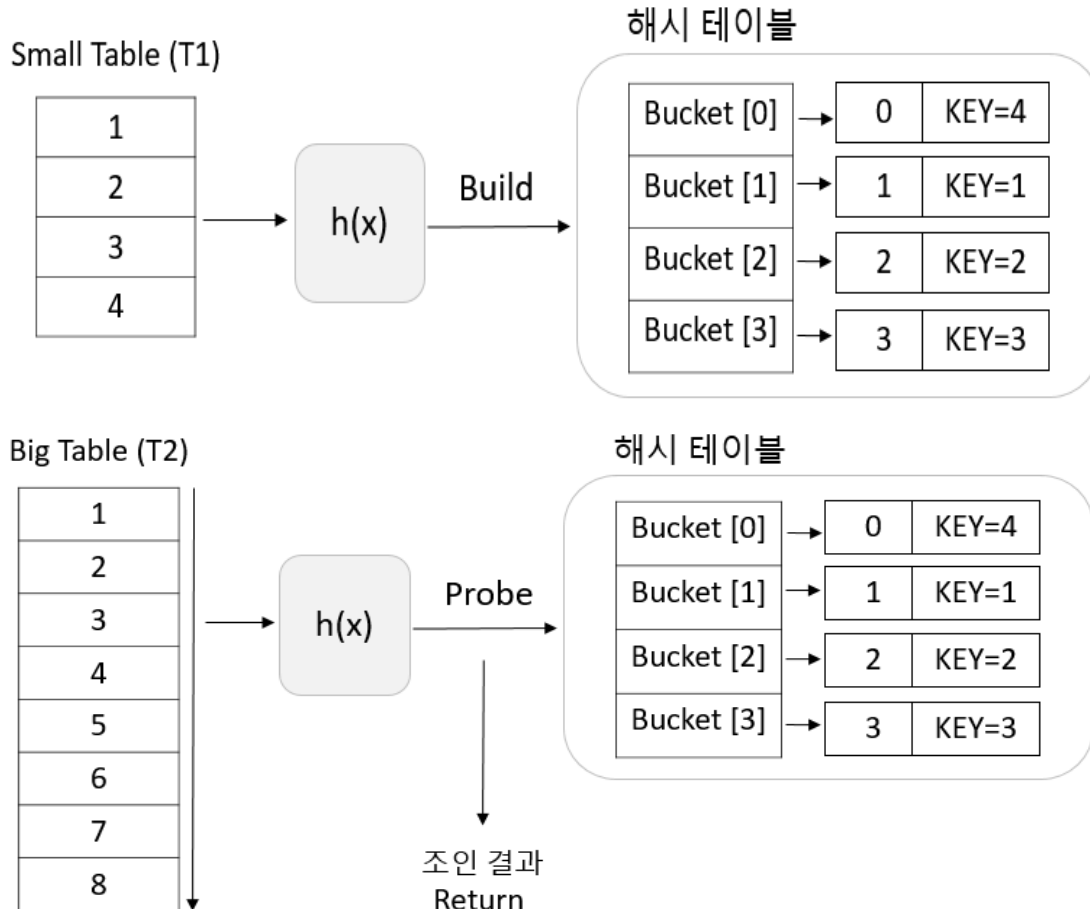
Execution time: 0.064 ms

해시 조인의 기본 원리

- 해시 조인은 해시 함수를 이용한다. 해시 함수(h)는 다음과 같은 특성을 갖는다.
 1. $X=Y$ 이면 반드시 $h(X)=h(Y)$ 이다.
 2. $h(X) \neq h(Y)$ 이면 반드시 $X \neq Y$ 이다.
 3. $X \neq Y$ 이면 $h(X) \neq h(Y)$ 인 것이 가장 이상적이다.
 4. $X \neq Y$ 이면 $h(X)=h(Y)$ 일 수도 있다. 이것을 해시 충돌이라고 한다.

In-Memory 해시 조인

- In-Memory 해시 조인은 해시 Build 작업을 `work_mem` 공간 내에서 모두 처리할 수 있을 때 사용하는 방식이다.



In-Memory 해시 조인의 Explain 결과 예제

```
explain (costs false, analyze, buffers)
select *
from   t1 a, t2 b
where  a.c1 = b.c1;
```

QUERY PLAN

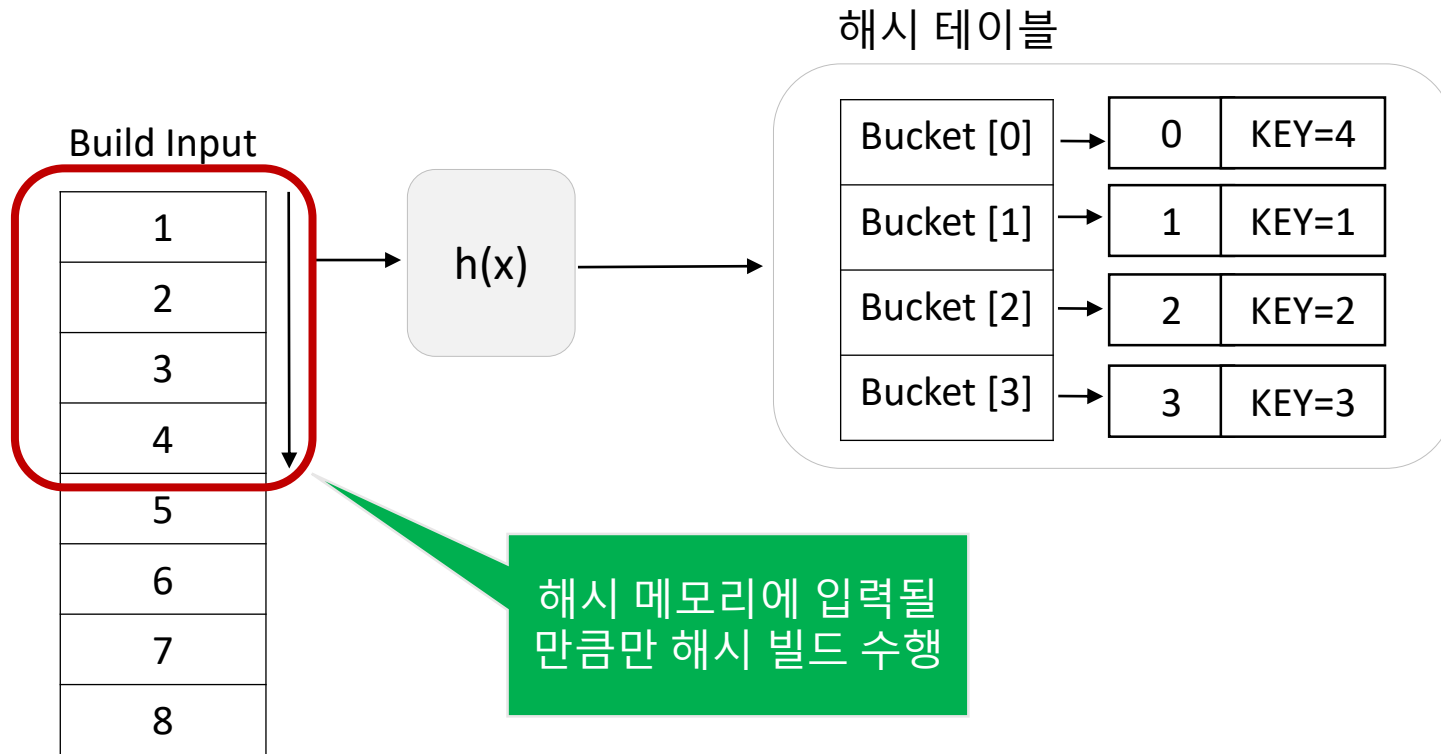
```
-----
Hash Join (actual time=0.131..13.996 rows=10 loops=1)
  Hash Cond: (b.c1 = a.c1)
  Buffers: shared hit=10
  -> Seq Scan on t2 b (actual time=0.006..0.211 rows=1000 loops=1)
        Buffers: shared hit=6
  -> Hash (actual time=0.090..0.090 rows=10 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        Buffers: shared hit=1
        -> Seq Scan on t1 a (actual time=0.001..0.004 rows=10 loops=1)
              Buffers: shared hit=1
```

In-Memory로 처리할 수 없을 때 사용되는 해시 조인 방식들

- 해시 Build 작업을 work_mem 공간 내에서 모두 처리할 수 없을 때 사용하는 방식은 크게 3가지이다.

해시 조인 유형	설명	단점
Classic 해시 조인	<ul style="list-style-type: none">가장 비효율적인 방식MariaDB에서 채용함 (Block Nested Loop Hash - BNLH)	<ul style="list-style-type: none">Probe 테이블을 n번 액세스
Grace 해시 조인	<ul style="list-style-type: none">Classic 해시 조인의 문제점을 개선	<ul style="list-style-type: none">0번 파티션도 Temp IO 발생
Hybrid 해시 조인	<ul style="list-style-type: none">Grace 해시 조인의 단점을 개선PostgreSQL에서 채용함 (Skew 데이터 처리 로직도 포함)	<ul style="list-style-type: none">없음

Classic 해시 조인 수행 절차 (Build Phase #1)



Classic 해시 조인 수행 절차 (Probe Phase #1)

Probe 테이블

1
2
3
4
5
6
7
8

$h(x)$

Probe

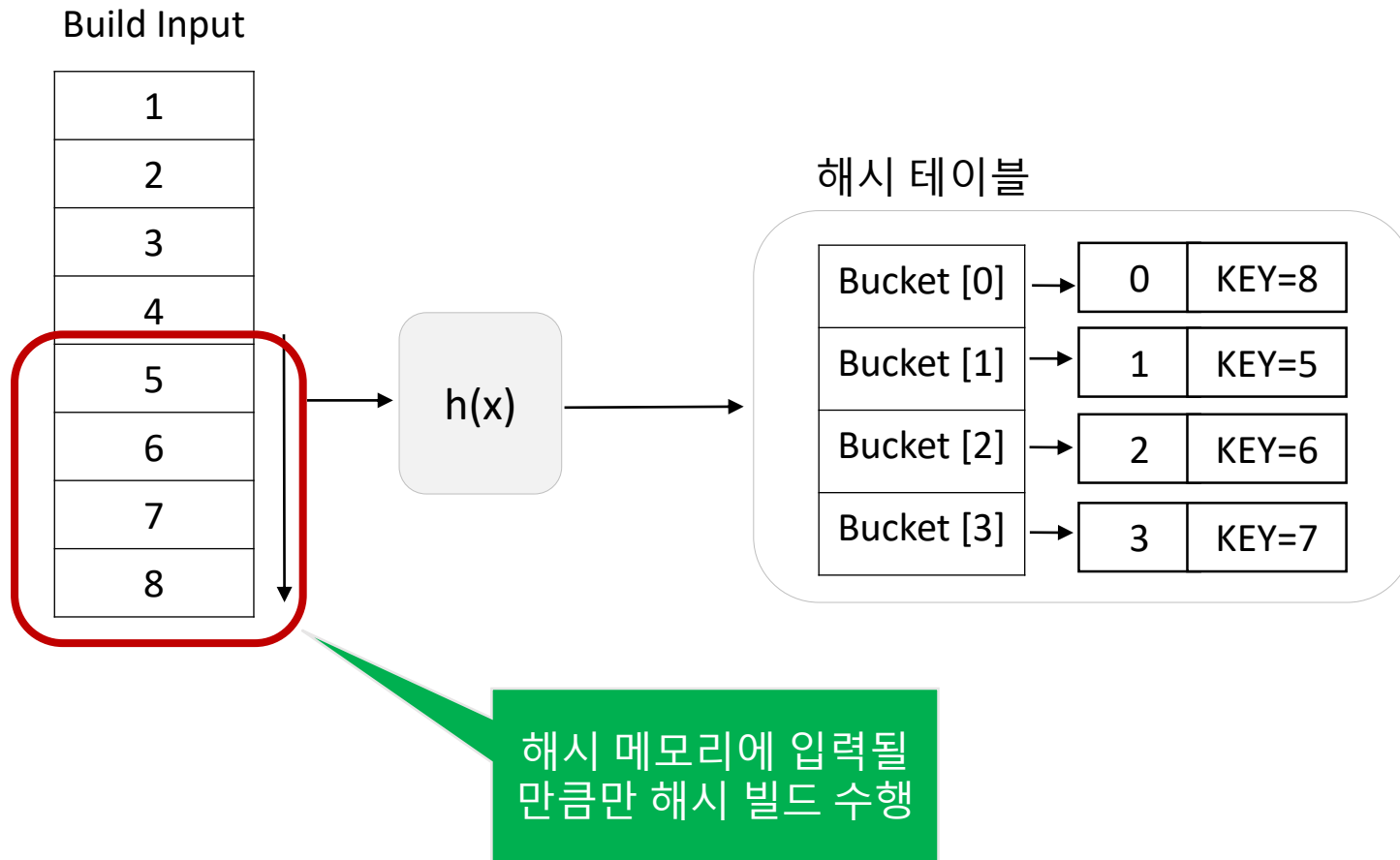
해시 테이블

Bucket [0]	→	0	KEY=4
Bucket [1]	→	1	KEY=1
Bucket [2]	→	2	KEY=2
Bucket [3]	→	3	KEY=3

조인 결과
Return

Probe 테이블 액세스
(1회)

Classic 해시 조인 수행 절차 (Build Phase #1)



Classic 해시 조인 수행 절차 (Probe Phase #2)

Probe 테이블

1
2
3
4
5
6
7
8

$h(x)$

Probe

해시 테이블

Bucket [0]	→	0	KEY=4
Bucket [1]	→	1	KEY=1
Bucket [2]	→	2	KEY=2
Bucket [3]	→	3	KEY=3

조인 결과
Return

Probe 테이블 액세스
(2회)

Grace 해시 조인 수행 절차 (Build Phase)

Build Input

1
2
3
4
5
6
7
8

$h1(x)$

Temporary 테이블스페이스

Build 파티션#0

4	8
---	---

Build 파티션#1

1	5
---	---

Build 파티션#2

2	6
---	---

Build 파티션#3

3	7
---	---

Grace 해시 조인 수행 절차 (Probe Phase)

Probe Input

1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8

$h1(x)$

Temporary 테이블스페이스

Probe, Build 파티션
Pair 끼리
해시 조인 수행

Probe 파티션#0

4	4	8	8
---	---	---	---

Build 파티션#0

4	8
---	---

Probe 파티션#1

1	1	5	5
---	---	---	---

Build 파티션#1

1	5
---	---

Probe 파티션#2

2	2	6	6
---	---	---	---

Build 파티션#2

2	6
---	---

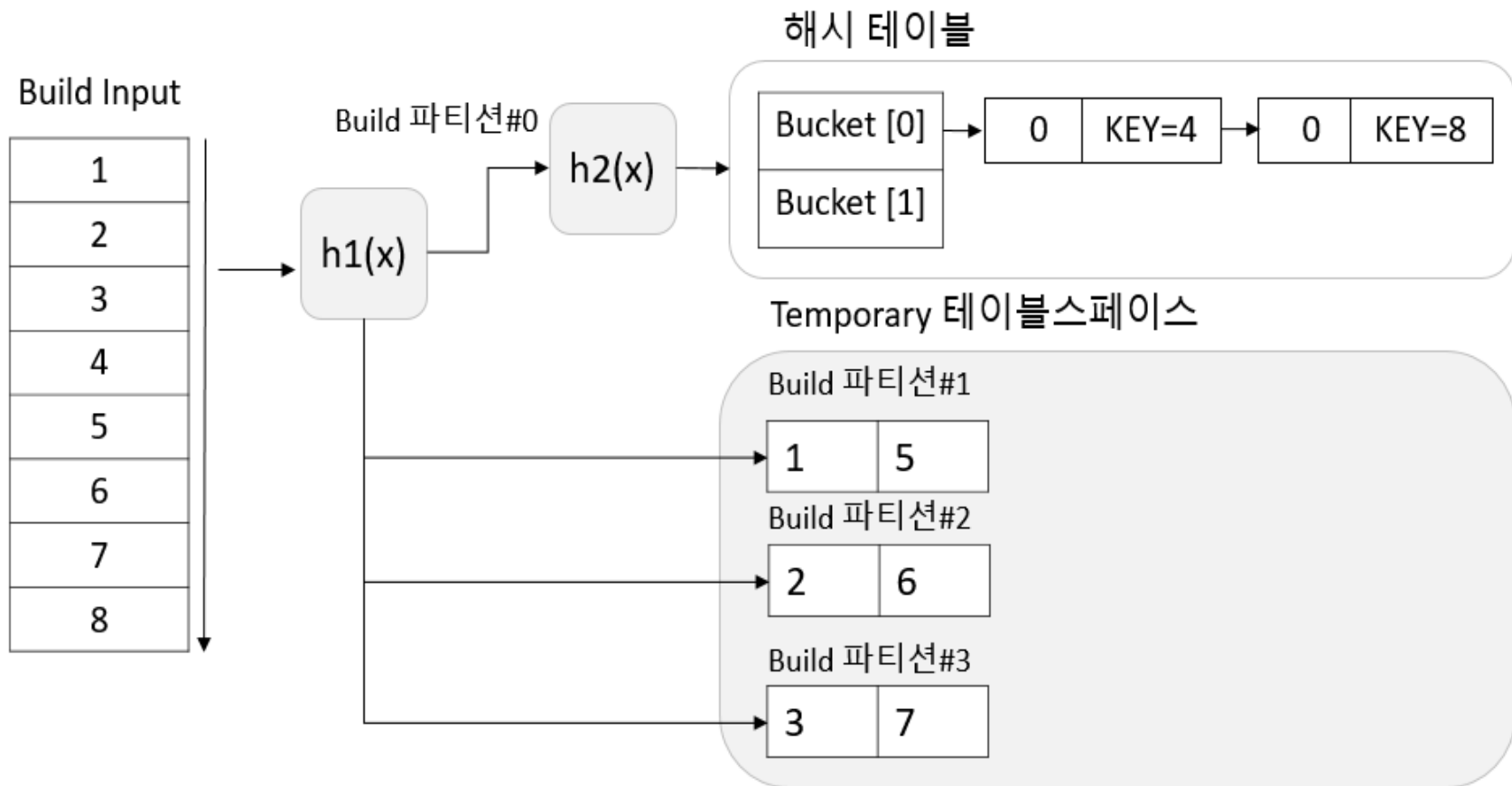
Probe 파티션#3

3	3	7	7
---	---	---	---

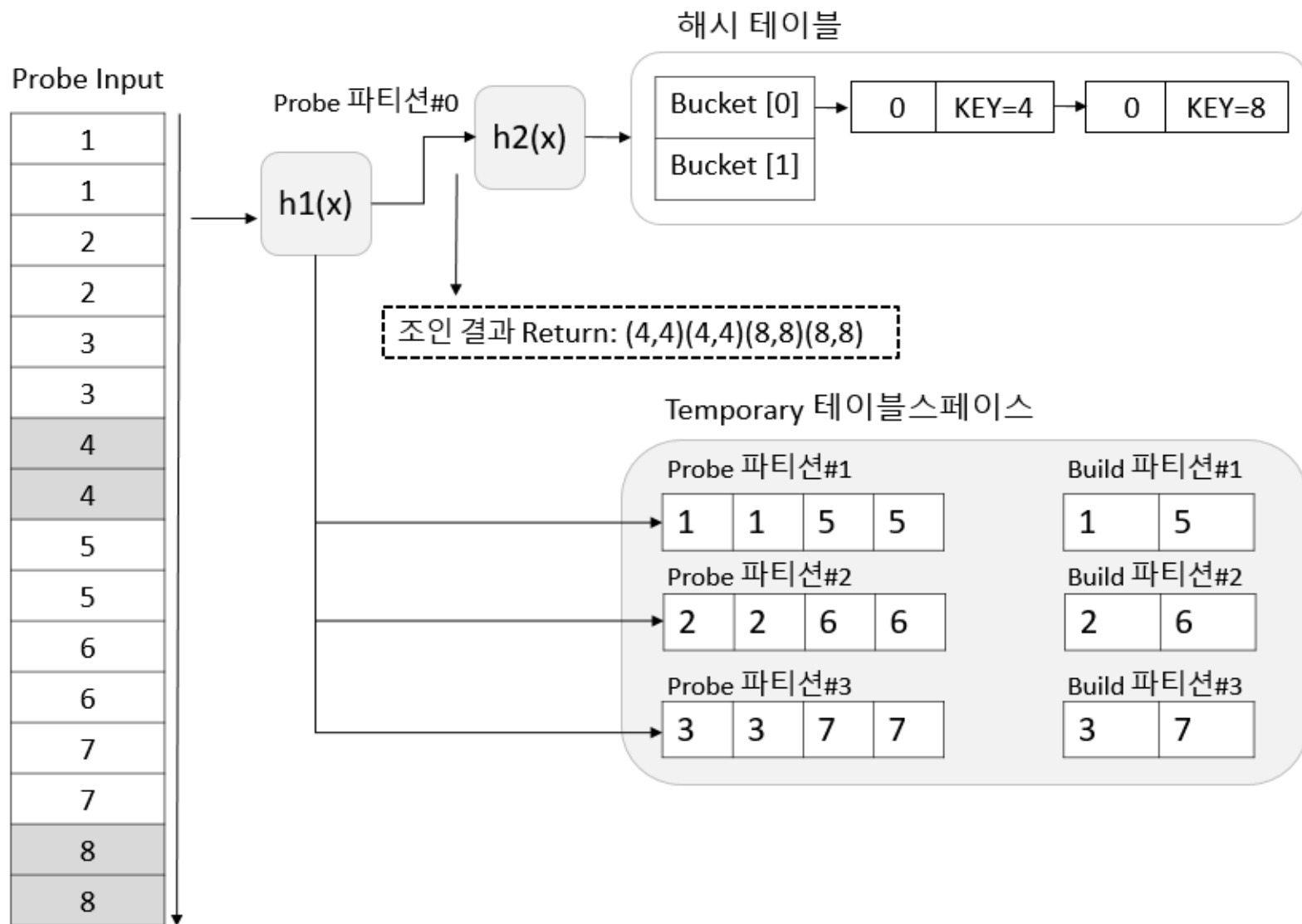
Build 파티션#3

3	7
---	---

Hybrid 해시 조인 수행 절차 (Build Phase)



Hybrid 해시 조인 수행 절차 (Probe Phase)



Hybrid 해시 조인의 Explain 결과 예제

```
explain (costs false, analyze, buffers)
select *
from   t1 a, t2 b
where  a.c1 = b.c1;
```

QUERY PLAN

```
-----
Hash Join (actual time=9988.924..29263.532 rows=400000 loops=1)
  Hash Cond: (b.c1 = a.c1)
  Buffers: shared hit=2704, temp read=1889 written=1875
  -> Seq Scan on t2 b (actual time=0.007..67.202 rows=400000 loops=1)
      Buffers: shared hit=2163
  -> Hash (actual time=9988.600..9988.600 rows=100000 loops=1)
      Buckets: 32768 Batches: 8 Memory Usage: 838kB
      Buffers: shared hit=541, temp written=371
      -> Seq Scan on t1 a (actual time=0.006..25.134 rows=100000 loops=1)
          Buffers: shared hit=541
```


Outer Join

Outer 조인 개요

- Outer 조인은 조인 성공 여부와 무관하게 기준 집합의 레코드를 모두 출력하는 방식이다.
- PostgreSQL은 ANSI-SQL 형태의 Outer 조인 문법만을 지원한다.
- Outer 조인은 결과 건수에 영향을 미치므로 사용시에 주의해야 한다.
- NL Outer 조인은 기준 집합을 항상 먼저 Driving 한다.
- Hash Outer 조인은 SWAP INPUT 기능을 제공한다.

간단한 퀴즈로 풀어보는 Outer 조인의 속성

```
create table o1 (c1 integer, dummy char(10));
create table o2 (c1 integer, dummy char(10));

insert into o1 values(1, 'dummy');
insert into o1 values(2, 'dummy');

insert into o2 values(2, 'dummy');
insert into o2 values(3, 'dummy');
```

간단한 쿼리로 풀어보는 Outer 조인의 속성

쿼리1)

```
select a.c1, b.c1
from   o1 a LEFT JOIN o2 b ON a.c1=b.c1;
```

c1	c1
1	
2	2

쿼리2)

```
select a.c1, b.c1
from   o1 a LEFT JOIN o2 b ON a.c1=b.c1 AND a.c1=1;
```

c1	c1
1	
2	

쿼리3)

```
select a.c1, b.c1
from   o1 a LEFT JOIN o2 b ON a.c1=b.c1 AND a.c1=2;
```

c1	c1
1	
2	2

ON 절의 조건은
조인 수행 여부와
관련 있다.

간단한 쿼리로 풀어보는 Outer 조인의 속성

쿼리4)

```
select a.c1, b.c1
from   o1 a LEFT JOIN o2 b ON a.c1=b.c1
WHERE a.c1=1;
```

c1		c1
1		

WHERE 절의 조건은
출력 결과와 관련 있다.
즉, 결과를 필터링한다.

쿼리5)

```
select a.c1, b.c1
from   o1 a LEFT JOIN o2 b ON a.c1=b.c1
WHERE a.c1=2;
```

c1		c1
2		2

테스트 환경 구성

```
drop table t1;
drop table t2;

create table t1 (c1 integer, c2 integer, dummy char(10));
create table t2 (c1 integer, c2 integer, dummy char(10));

insert into t1 select i,i, 'dummy' from generate_series(1,1000) a(i);
insert into t2 select i,i, 'dummy' from generate_series(1,10000000) a(i);

create index t1_idx01 on t1(c1);
create index t2_idx01 on t2(c1);

create unique index t2_uk on t2(c2);

analyze t1;
analyze t2;
```

일반 조인의 예

```
set enable_hashjoin=off;  
set enable_mergejoin=off;  
set enable_bitmapscan=off;
```

```
explain (costs false, analyze, buffers)  
select * from t1 a, t2 b  
where  a.c1 between 1 and 10  
and    a.c1 = b.c1  
and    b.c2 = 1;
```

QUERY PLAN

Nested Loop (actual time=0.013..0.014 rows=1 loops=1)

Buffers: shared hit=8

-> **Index Scan using t2_uk on t2 b**

Index Cond: (c2 = 1)

Buffers: shared hit=4

-> **Index Scan using t1_idx01 on t1 a**

Index Cond: ((c1 = b.c1) AND (c1 >= 1) AND (c1 <= 10))

Buffers: shared hit=4

Planning time: 0.212 ms

Execution time: 0.036 ms

일반 조인인 경우에는
T2 테이블을 Driving 한다.

Outer NL 조인의 예

```
set enable_material=off;
```

```
explain (costs false, analyze, buffers)
```

```
select * from t1 a LEFT JOIN t2 b ON a.c1=b.c1 AND b.c2=1  
where a.c1 between 1 and 10;
```

QUERY PLAN

Nested Loop Left Join (actual time=0.014..0.033 rows=10 loops=1)

Join Filter: (a.c1 = b.c1)

Rows Removed by Join Filter: 9

Buffers: shared hit=44

-> **Index Scan using t1_idx01 on t1 a**

Index Cond: ((c1 >= 1) AND (c1 <= 10))

Buffers: shared hit=4

-> **Index Scan using t2_uk on t2 b**

Index Cond: (c2 = 1)

Buffers: shared hit=40

Planning time: 0.127 ms

Execution time: 0.050 ms

NL Outer 조인이므로
기준집합을 Driving 한다.

Hash Outer 조인의 예 (기준 집합이 작은 경우)

```
set enable_nestloop=off;  
set enable_mergejoin=off;  
set enable_hashjoin=on;
```

```
explain (costs false, analyze, buffers)  
select * from t1 a LEFT JOIN t2 b ON a.c1=b.c1;  
QUERY PLAN
```

```
-----  
Hash Right Join (actual time=0.323..2970.618 rows=1000 loops=1)  
  Hash Cond: (b.c1 = a.c1)  
  Buffers: shared hit=63702  
    -> Seq Scan on t2 b (actual time=0.003..1354.998 rows=10000000 loops=1)  
        Buffers: shared hit=63695  
    -> Hash (actual time=0.313..0.313 rows=1000 loops=1)  
        Buckets: 1024  Batches: 1  Memory Usage: 58kB  
        Buffers: shared hit=7  
      -> Seq Scan on t1 a (actual time=0.011..0.154 rows=1000 loops=1)  
          Buffers: shared hit=7  
Planning time: 0.115 ms  
Execution time: 2970.823 ms
```

기준 집합이 Build 테이블이 된다.

Hash Outer 조인의 예 (기준 집합이 큰 경우)

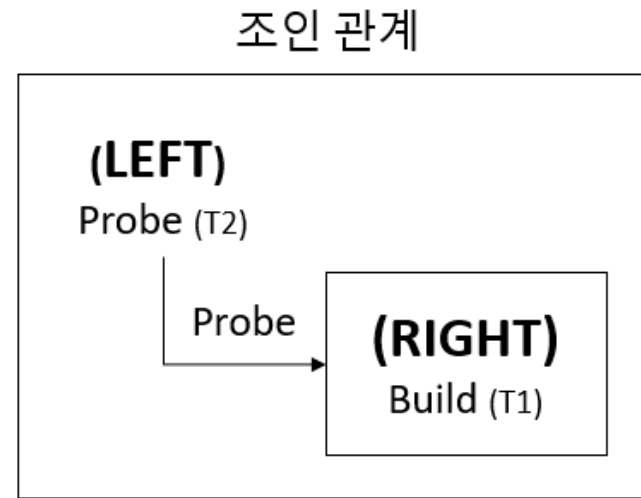
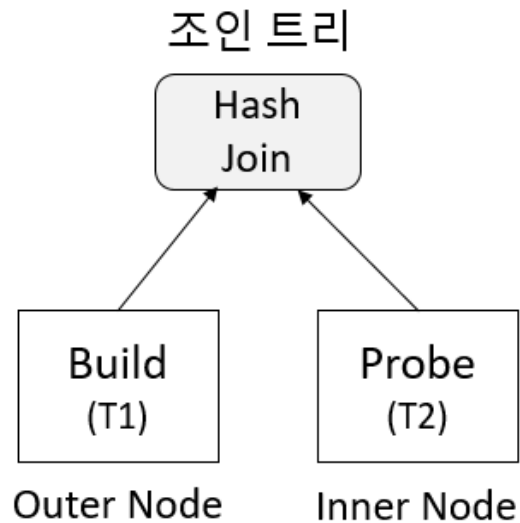
```
explain (costs false, analyze, buffers)
select * from t2 b LEFT JOIN t1 a ON a.c1=b.c1;
```

QUERY PLAN

```
-----
Hash Left Join (actual time=0.411..4.923 rows=10000000 loops=1)
  Hash Cond: (b.c1 = a.c1)
  Buffers: shared hit=63702
  -> Seq Scan on t2 b (actual time=0.011..1388.394 rows=10000000 loops=1)
    Buffers: shared hit=63695
  -> Hash (actual time=0.394..0.394 rows=1000 loops=1)
    Buckets: 1024 Batches: 1 Memory Usage: 58kB
    Buffers: shared hit=7
    -> Seq Scan on t1 a (actual time=0.006..0.200 rows=1000 loops=1)
      Buffers: shared hit=7
Planning time: 0.096 ms
Execution time: 5405.386 ms
```

기준 집합이 Probe 테이블이 된다.
즉, 작은 테이블인 T1을 Build
테이블로 선택한다. 이것을 Build
Input SWAP 이라고 한다.

Hash Right Join, Hash Left Join의 의미



- Explain 결과의 Hash Right Join은 기준 테이블이 RIGHT, 즉 Build 테이블이라는 사실을 알려준다.
- Explain 결과의 Hash Left Join은 기준 테이블이 LEFT, 즉 Probe 테이블이라는 사실을 알려준다.

Query Rewrite

Query Rewrite란?

- 사용자가 작성한 쿼리를 Optimizer가 더 좋은 실행계획을 수립할 수 있는 형태로 변경하는 것을 Query Rewrite라고 한다.
 - 1) 서브 쿼리 Collapse : 서브 쿼리를 Main 쿼리에 병합하는 기법
 - 2) View Merging : 뷰 또는 인라인 뷰를 풀어헤쳐 테이블 간의 조인으로 변경하는 방법
 - 3) JPPD : View Merging이 실패한 경우, 조인 조건을 뷰 내부로 밀어 넣는 방법

서브쿼리 Collapse 테스트를 위한 환경 설정

```
drop table t1;
drop table t2;

create table t1 (c1 integer, c2 integer, dummy char(100));
create table t2 (c1 integer, c2 integer, dummy char(100));

insert into t1 select i, i, 'dummy' from generate_series(1,10000) a(i);
insert into t2 select mod(i,10000), i, 'dummy' from generate_series(1,1000000)
a(i);

analyze t1;
analyze t2;
```

서브쿼리 Collapse 발생 시의 Explain 결과

```
explain (costs false, analyze, buffers)
select a.*
from t1 a
where a.c2 between 9001 and 10000
and EXISTS (select 1 from t2 b where b.c1=a.c1);
QUERY PLAN
```

서브 쿼리 테이블 T2를
Group by 한 후에 Hash
조인으로 수행됨. 즉, 서브
쿼리 Collapse가 발생함

```
Hash Join (actual time=380.250..388.740 rows=999 loops=1)
  Hash Cond: (b.c1 = a.c1)
  Buffers: shared hit=17415
  -> HashAggregate (actual time=378.056..384.118 rows=10000 loops=1)
    Group Key: b.c1
    Buffers: shared hit=17242
    -> Seq Scan on t2 b (actual time=0.007..145.752 rows=1000000...)
      Buffers: shared hit=17242
  -> Hash (actual time=2.168..2.168 rows=1000 loops=1)
    Buckets: 1024 Batches: 1 Memory Usage: 146kB
    Buffers: shared hit=173
    -> Seq Scan on t1 a (actual time=1.493..1.820 rows=1000 loops=1)
      Filter: ((c2 >= 9001) AND (c2 <= 10000))
      Rows Removed by Filter: 9000
      Buffers: shared hit=173
Planning time: 0.189 ms
Execution time: 388.923 ms
```

서브쿼리 Collapse를 억제하는 방법

- OFFSET 0을 적용하면 Filter방식으로 유도된다.

```
explain (costs false, analyze, buffers)
select a.*
from   t1 a
where  a.c2 between 9001 and 10000
and    EXISTS (select 1 from t2 b where b.c1=a.c1 OFFSET 0);
```

QUERY PLAN

```
-----
Seq Scan on t1 a (actual time=1.747..961.744 rows=999 loops=1)
  Filter: ((c2 >= 9001) AND (c2 <= 10000) AND (SubPlan 1))
  Rows Removed by Filter: 9001
  Buffers: shared hit=181538
SubPlan 1
  -> Seq Scan on t2 b (actual time=0.957..0.957 rows=1 loops=1000)
      Filter: (c1 = a.c1)
      Rows Removed by Filter: 10490
      Buffers: shared hit=181365
```

Planning time: 0.064 ms

Execution time: 962.038 ms

SeqScan 1000번 수행

인덱스 생성 후에는 Nested Loop Semi Join으로 수행된다.

```
create index t2_idx01 on t2(c1);
explain (costs false, analyze, buffers)
select a.*
from   t1 a
where  a.c2 between 9001 and 10000
and    EXISTS (select 1 from t2 b where b.c1=a.c1);
```

QUERY PLAN

```
Nested Loop Semi Join (actual time=1.048..7.071 rows=999 loops=1)
  Buffers: shared hit=3903 read=274
  -> Seq Scan on t1 a (actual time=1.005..1.301 rows=1000 loops=1)
      Filter: ((c2 >= 9001) AND (c2 <= 10000))
      Rows Removed by Filter: 9000
      Buffers: shared hit=173
  -> Index Only Scan using t2_idx01 on t2 b
      (actual time=0.005..0.005 rows=1 loops=1000)
      Index Cond: (c1 = a.c1)
      Heap Fetches: 999
      Buffers: shared hit=3730 read=274
Planning time: 0.352 ms
Execution time: 7.178 ms
```

View Merging 개요

- 뷰를 풀어헤쳐서 테이블 간의 조인으로 변경함으로써 , 다양한 조인 순서와 조인 방법을 선택할 수 있다. 이때 View는 2개로 구분된다.
- Simple View는 항상 View Merging에 성공한다.
- Complex View는 항상 View Merging에 실패한다.

- 1) Simple View: Group by , Distinct와 같은 Aggregate를 사용하지 않는 뷰
- 2) Complex View: Group by , Distinct와 같은 Aggregate를 사용하는 뷰

테스트 환경 구성

```
drop table customer;
drop table sales;
create table customer (custid integer, custname varchar(30), dummy char(10));
create table sales (salesid integer, custid integer, salesdate date, dummy
char(10));

insert into customer select i, 'ksy'||i::varchar, 'dummy' from
generate_series(1,100000) a(i);

do $$
begin
  for i in 1..10000 loop
    for j in 1..100 loop
      insert into sales values (j+(10000*(i-1)), mod(j+(10000*(i-1)),100000),
to_date('20170101','YYYYMMDD')+j-1, 'dummy');
    end loop;
  end loop;
end$$;

create index customer_idx01 on customer(custid);
create index customer_idx02 on customer(custname);
create index sales_idx01 on sales(salesid);
create index sales_idx02 on sales(custid);

analyze customer;
analyze sales;
```

PostgreSQL은 Simple View Merging을 지원한다.

```
explain (costs false, analyze, buffers)
select *
from   (select * from customer a where a.custname = 'ksy90001') a,
       (select * from sales      b where b.salesdate between DATE '20170101'
                                              and DATE '20170331') b
where  a.custid = b.custid;
```

QUERY PLAN

```
-----
Nested Loop (actual time=0.270..1.894 rows=1000 loops=1)
  Buffers: shared hit=1009
    -> Index Scan using customer_idx02 on customer a
        Index Cond: ((custname)::text = 'ksy90001'::text)
        Buffers: shared hit=4
    -> Bitmap Heap Scan on sales b
        Recheck Cond: (custid = a.custid)
        Filter: ((salesdate >= '2017-01-01'::date) AND
                  (salesdate <= '2017-03-31'::date))
        Heap Blocks: exact=1000
        Buffers: shared hit=1005
        -> Bitmap Index Scan on sales_idx02
            Index Cond: (custid = a.custid)
            Buffers: shared hit=5
Planning time: 0.232 ms
Execution time: 2.044 ms
```

즉, Simple View는 다음과 같이 View Merging 된다.

```
select *  
from    customer a, sales b  
where   a.custid    = b.custid  
and     a.custname  = 'ksy90001'  
and     b.salesdate between DATE '20170101'  
                                and DATE '20170331';
```

(참고)

너무나 당연해 보이는 기능이지만, MySQL은 5.7부터 Simple View Merging 기능을 제공한다.

Complex View인 경우를 살펴보자.

퀴즈) 아래의 쿼리는 어떻게 수행될까?

- ① 조인을 수행한 후에 Group by 를 수행한다. 즉, Complex View Merging을 한다.
- ② a.custname='ksy90001'에 해당되는 b.custid 상수 조건을 View 내에 밀어 넣는다. 즉, JPPD (Join Predicate Push-Down)를 수행한다.
- ③ Complex View Merging 뿐 아니라 JPPD도 수행하지 못한다.

```
set enable_mergejoin=off;

explain (costs false, analyze, buffers)
select a.*, b.*
from   customer a,
       (select count(*) cnt, custid, salesdate
        from   sales
        group by custid, salesdate) b
where  a.custid=b.custid
and    a.custname = 'ksy90001';
```

Complex View는 View Merging이 실패한다.

QUERY PLAN

```
-----  
Hash Join (actual time=1075.692..1118.784 rows=1 loops=1)  
  Hash Cond: (sales.custid = a.custid)  
  Buffers: shared hit=6374, temp read=3070 written=3070  
  -> GroupAggregate (actual time=645.246..1118.299 rows=1000 loops=1)  
    Group Key: sales.custid, sales.salesdate  
    Buffers: shared hit=6370, temp read=3070 written=3070  
    -> Sort (actual time=644.662..944.562 rows=1000000 loops=1)  
      Sort Key: sales.custid, sales.salesdate  
      Sort Method: external merge  Disk: 17568kB  
      Buffers: shared hit=6370, temp read=3070 written=3070  
      -> Seq Scan on sales (actual time=0.006..167.947 rows=1000000)  
        Buffers: shared hit=6370  
    -> Hash (actual time=0.016..0.016 rows=1 loops=1)  
      Buckets: 1024  Batches: 1  Memory Usage: 32kB  
      Buffers: shared hit=4  
      -> Index Scan using customer_idx02 on customer  
        time=0.014..0.014 rows=1 loops=1  
        Index Cond: ((custname)::text = 'ksy90001'::text)  
        Buffers: shared hit=4  
Planning time: 0.124 ms  
Execution time: 1121.979 ms
```

Sales 테이블을 Seq Scan 한다.

어떻게 해야 할까?

2가지 방법을 이용해서 JPPD를 동작하게 할 수 있다.

- ① 조인 조건을 상수화 한다.
- ② LATERAL View를 사용한다.

조인 조건을 상수화해서 JPPD를 구현

```
select custid from customer where custname = 'ksy90001';
custid
-----
90001
```

```
explain (costs false, analyze, buffers)
select a.*, b.*
from   customer a,
       (select count(*) cnt, custid, salesdate
        from   sales
        group by custid, salesdate) b
where  a.custid=b.custid
and    a.custid in (90001);
```

조인 조건을 상수화해서 JPPD를 구현

QUERY PLAN

```
-----  
Nested Loop (actual time=1.632..1.633 rows=1 loops=1)  
  Buffers: shared hit=1008  
    -> Index Scan using customer_idx01 on customer a  
        Index Cond: (custid = 90001)  
        Buffers: shared hit=3  
    -> HashAggregate (actual time=1.623..1.623 rows=1 loops=1)  
        Group Key: sales.custid, sales.salesdate  
        Buffers: shared hit=1005  
      -> Bitmap Heap Scan on sales (actual time=0.227..1.386 rows=1000)  
          Recheck Cond: (custid = 90001)  
          Heap Blocks: exact=1000  
          Buffers: shared hit=1005  
        -> Bitmap Index Scan on sales_idx02  
            Index Cond: (custid = 90001)  
            Buffers: shared hit=5  
Planning time: 0.152 ms  
Execution time: 1.669 ms
```

상수 조건을 이용해서 Index Scan을 수행한다.

LATERAL View를 이용해서 JPPD를 구현

- PostgreSQL은 9.3 버전부터 LATERAL View를 제공한다.
- 이는 JPPD를 직접 구현할 수 있는 매우 획기적인 튜닝 방법이다.
- ORACLE도 12c부터는 Lateral View를 사용자에게 제공한다.

```
explain (costs false, timing false, analyze)
select a.*, b.*
from   customer a,
       LATERAL (select count(*) cnt, custid, salesdate
                 from   sales
                 where  sales.custid = a.custid
                 group by custid, salesdate) b
where  a.custname = 'ksy90001';
```

측면 (LATERAL) 테이블인 a의
조건을 입력한다.

LATERAL View를 이용해서 JPPD를 구현

QUERY PLAN

Nested Loop (actual rows=1 loops=1)

-> Index Scan using customer_idx02 on customer a (actual rows=1 loops=1)
Index Cond: ((custname)::text = 'ksy90001'::text)

-> HashAggregate (actual rows=1 loops=1)

Group Key: sales.custid, sales.salesdate

-> Bitmap Heap Scan on sales (actual rows=1000 loops=1)

Recheck Cond: (custid = a.custid)

Heap Blocks: exact=1000

-> **Bitmap Index Scan on sales_idx02** (actual rows=1000 loops=1)
Index Cond: (custid = a.custid)

Planning time: 0.112 ms

Execution time: 1.449 ms

LATERAL View는 LEFT JOIN도 지원한다.

- LEFT JOIN시에는 ON TRUE 키워드를 이용한다.

```
explain (costs false, timing false, analyze)
select a.*, b.*
from   customer a
       LEFT JOIN
       LATERAL (select count(*) cnt, custid, salesdate
                from   sales
                where  sales.custid = a.custid
                group by custid, salesdate) b ON TRUE
where  a.custname = 'ksy90001';
```

QUERY PLAN

```
-----
Nested Loop Left Join (actual rows=1 loops=1)
->  Index Scan using customer_idx02 on customer a (actual rows=1 loops=1)
    Index Cond: ((custname)::text = 'ksy90001'::text)
->  HashAggregate (actual rows=1 loops=1)
    Group Key: sales.custid, sales.salesdate
    ->  Bitmap Heap Scan on sales (actual rows=1000 loops=1)
        Recheck Cond: (custid = a.custid)
        Heap Blocks: exact=1000
        ->  Bitmap Index Scan on sales_idx02 (actual rows=1000 loops=1)
            Index Cond: (custid = a.custid)

Planning time: 0.161 ms
Execution time: 2.080 ms
```

PG_HINT_PLAN

힌트의 필요성과 PG_HINT_PLAN 개요

- 힌트는 필요할까? (논의해보자)
- 특정 인덱스를 지정하거나, 조인 순서를 지정하거나, 여러 개의 테이블 간의 조인 방법을 각 조인마다 다르게 선택하는 것을 사용자가 지정할 수 있다면, 이는 매우 강력한 무기일 것이다.
- 하지만 지금까지 학습한 방법으로는 이 모든 것을 해결할 수 없다.
- ORACLE은 Hint를 이용해서 이러한 작업을 수행할 수 있다.
- 그런데 PostgreSQL은 Hint를 제공하지 않는다. (향후 계획에도 포함되지 않았다)
- 어떻게 할 것인가?
- 다행히 PG_HINT_PLAN 이란 기능이 제공된다.
- 이는 Hint 가 아니라 Plan Tree 자체를 변경하는 기법이다.
- 따라서 Hint와 달리 Optimizer는 PG_HINT_PLAN을 무시할 수 없다.
- 즉, 매우 강력하면서도 위험한 무기인 셈이다.

PG_HINT_PLAN 설치

```
-- 다운로드
https://osdn.net/projects/pghintplan/

-- ROOT 유저로 설치 진행
# gunzip pg_hint_plan96-1.1.3.tar.gz
# tar xvf pg_hint_plan96-1.1.3.tar
# make PG_CONFIG=/usr/local/pgsql/bin/pg_config
# make PG_CONFIG=/usr/local/pgsql/bin/pg_config install

-- postgresql.conf에 등록
shared_preload_libraries = 'pg_hint_plan'

-- 또는 개별 세션에서 LOAD
postgres=# LOAD 'pg_hint_plan';
```


PG_HINT_PLAN이 제공하는 Hint들

- 액세스 방법

사용법	설명
SeqScan (table)	Seq Scan 방식으로 유도한다.
IndexScan (table index)	Index Scan 방식으로 유도한다.
IndexOnlyScan (table index)	Index Only Scan 방식으로 유도한다. 만일 Index Only Scan을 사용할 수 없다면, Index Scan 방식을 사용한다.
BitmapScan (table index)	Bitmap Scan 방식으로 유도한다.
NoIndexScan (table)	Index Scan과 Index Only Scan 방식을 사용하지 않도록 한다.
NoIndexOnlyScan (table)	Index Only Scan 방식을 사용하지 않도록 한다.
NoBitmapScan (table)	Bitmap Scan 방식을 사용하지 않도록 한다.

PG_HINT_PLAN이 제공하는 Hint들

- 조인 방법

사용법	설명
NestLoop (table table)	NL 조인으로 유도한다.
HashJoin (table table)	해시 조인으로 유도한다.
MergeJoin (table table)	Merge 조인으로 유도한다.
NoNestLoop (table table)	NL 조인을 사용하지 않도록 한다.
NoHashJoin (table table)	해시 조인을 사용하지 않도록 한다
NoMergeJoin (table table)	Merge 조인을 사용하지 않도록 한다

PG_HINT_PLAN이 제공하는 Hint들

- 조인 순서

사용법	설명
Leading (table table [table...])	조인 순서를 제어한다. 단, 조인 방향을 제어하지는 않는다.
Leading ((table table))	조인 순서와 방향을 제어한다. 즉, 왼쪽 테이블이 Driving 또는 Outer 테이블이다.

테스트 환경 구성

```
drop table t1;
drop table t2;
drop table t3;
drop table t4;

create table t1 (c1 int, c2 int, c3 int, dummy char(100));
create index t1_idx1 on t1 (c1, c2, c3);
create index t1_idx2 on t1 (c2, c3);
create index t1_idx3 on t1 (c3);

create table t2 (c1 int, c2 int, c3 int, dummy char(100));
create index t2_idx1 on t2 (c1, c2, c3);
create index t2_idx2 on t2 (c2, c3);
create index t2_idx3 on t2 (c3);

create table t3 (c1 int, c2 int, c3 int, dummy char(100));
create index t3_idx1 on t3 (c1, c2, c3);
create index t3_idx2 on t3 (c2, c3);
create index t3_idx3 on t3 (c3);

create table t4 (c1 int, c2 int, c3 int, dummy char(100));
create index t4_idx1 on t4 (c1, c2, c3);
create index t4_idx2 on t4 (c2, c3);
create index t4_idx3 on t4 (c3);
```

테스트 환경 구성

```
insert into t1 select 1, mod(i,100), mod(i,1000), 'dummy'
from generate_series(1,100000) a(i);

insert into t2 select 1, mod(i,100), mod(i,1000), 'dummy'
from generate_series(1,10000) a(i);

insert into t3 select 1, mod(i,100), i, 'dummy'
from generate_series(1,100) a(i);

insert into t4 select 1, mod(i,100), i, 'dummy'
from generate_series(1,100) a(i);

analyze t1;
analyze t2;
analyze t3;
analyze t4;
```

IndexScan Hint 테스트

- 옵티마이저의 판단 미스로 T1_IDX01 인덱스를 선택했다.

```
explain (costs false, analyze, buffers)
select a.*
from   t1 a
where  a.c1=1
and    a.c2 between 1 and 10
and    a.c3=1;
```

QUERY PLAN

Index Scan using t1_idx1 on t1 a (actual time=0.017..0.555 rows=100 loops=1)
Index Cond: ((c1 = 1) AND (c2 >= 1) AND (c2 <= 10) AND (c3 = 1))
Buffers: shared hit=145
Planning time: 0.085 ms
Execution time: 0.587 ms

IndexScan Hint 테스트

- 가장 효율이 좋은 T1_IDX3 인덱스를 이용하도록 유도한다.

```
load 'pg_hint_plan';
```

```
/*+ IndexScan(a t1_idx3) */
```

```
explain (costs false, analyze, buffers)
```

```
select a.*
```

```
from   t1 a
```

```
where  a.c1=1
```

```
and     a.c2 between 1 and 10
```

```
and     a.c3=1;
```

QUERY PLAN

Index Scan using t1_idx3 on t1 a (actual time=0.015..0.116 rows=100 loops=1)

Index Cond: (c3 = 1)

Filter: ((c2 >= 1) AND (c2 <= 10) AND (c1 = 1))

Buffers: shared hit=102

Planning time: 0.104 ms

Execution time: 0.145 ms

NestLoop 힌트 테스트

- 현재로서는 해시 조인이 가장 좋은 선택이다. 이를 Nest Loop 조인으로 변경해보자.

```
explain (costs false, analyze)
select a.*, b.*
from   t1 a, t2 b
where  a.c1=b.c1
and    a.c2=b.c2
and    a.c3=b.c3;
```

QUERY PLAN

```
-----
Hash Join (actual time=134.053..481.920 rows=1000000 loops=1)
  Hash Cond: ((a.c1 = b.c1) AND (a.c2 = b.c2) AND (a.c3 = b.c3))
  -> Seq Scan on t1 a (actual time=0.058..27.906 rows=100000 loops=1)
  -> Hash (actual time=133.885..133.885 rows=10000 loops=1)
        Buckets: 16384  Batches: 1  Memory Usage: 1545kB
        -> Seq Scan on t2 b (actual time=0.016..4.188 rows=10000 loops=1)
Planning time: 0.930 ms
Execution time: 570.457 ms
```


NestLoop 힌트 테스트

- 힌트 적용 후에 NL 조인으로 변경됐다. (결과적으로 더 느려졌지만, Hint가 적용된 것이다)
- 이때 Driving 테이블은 T2, Inner 테이블은 T1이다.
- T1을 Driving 테이블로 지정할 수 있을까?

```
load 'pg_hint_plan';
```

```
/*+ NestLoop(a b) */
```

```
explain (costs false, analyze)
```

```
select a.*, b.*
```

```
from   t1 a, t2 b
```

```
where  a.c1=b.c1
```

```
and    a.c2=b.c2
```

```
and    a.c3=b.c3;
```

QUERY PLAN

```
-----  
Nested Loop (actual time=0.100..800.554 rows=1000000 loops=1)  
  -> Seq Scan on t2 b (actual time=0.007..1.980 rows=10000 loops=1)  
    -> Index Scan using t1_idx2 on t1 a (actual time=0.005..0.057 rows=100  
loops=10000)  
      Index Cond: ((c2 = b.c2) AND (c3 = b.c3))  
      Filter: (b.c1 = c1)  
Planning time: 0.388 ms  
Execution time: 884.671 ms
```

Leading 힌트 테스트

- Leading 힌트를 적용했지만 여전히 T2 테이블이 Driving 테이블이다.
- 즉, Leading 힌트는 조인 순서만 정할 뿐, Direction은 지정하지 않는다.
- 예를 들어, Leading(a b c) 면 (a b)를 조인한 후에 (c)와 조인하라는 의미이다.

```
/*+ Leading(a b) NestLoop(a b) */
```

```
explain (costs false, analyze)
```

```
select a.*, b.*
```

```
from   t1 a, t2 b
```

```
where  a.c1=b.c1
```

```
and    a.c2=b.c2
```

```
and    a.c3=b.c3;
```

QUERY PLAN

```
-----  
Nested Loop (actual time=0.022..813.950 rows=1000000 loops=1)  
  -> Seq Scan on t2 b (actual time=0.007..1.927 rows=10000 loops=1)  
  -> Index Scan using t1_idx2 on t1 a (actual time=0.005..0.059 rows=100  
loops=10000)  
      Index Cond: ((c2 = b.c2) AND (c3 = b.c3))  
      Filter: (b.c1 = c1)  
Planning time: 0.240 ms  
Execution time: 896.697 ms
```

Leading() 힌트 테스트

- Direction까지 지정하려면 Leading() 힌트를 이용한다.
- 왼쪽 테이블이 Driving 또는 Probe 테이블이다.

```
/*+ Leading((a b)) NestLoop(a b) */
```

```
explain (costs false, analyze)
```

```
select a.*, b.*
```

```
from   t1 a, t2 b
```

```
where  a.c1=b.c1
```

```
and    a.c2=b.c2
```

```
and    a.c3=b.c3;
```

QUERY PLAN

Nested Loop (actual time=0.083..871.238 rows=1000000 loops=1)

-> **Seq Scan on t1 a** (actual time=0.007..15.878 rows=100000 loops=1)

-> **Index Scan using t2_idx2 on t2 b** (actual time=0.002..0.006 rows=10 loops=100000)

Index Cond: ((c2 = a.c2) AND (c3 = a.c3))

Filter: (a.c1 = c1)

Planning time: 0.237 ms

Execution time: 955.504 ms

Leading()와 조인 힌트로 조인 순서와 방법을 가지고 놀자!

- 아래의 SQL을 조인 순서는 T1-> T2-> T3 으로 조인 방법은 NL 조인으로 유도해보자.

```
/*+ Leading((a b) c) NestLoop(a b) NestLoop(a b c) */  
explain (costs false, analyze)  
select a.*, b.*, c.*  
from   t1 a, t2 b, t3 c  
where  a.c1=b.c1  
and    a.c2=b.c2  
and    a.c3=b.c3  
and    b.c1=c.c1  
and    b.c2=c.c2  
and    b.c3=c.c3;
```

Leading()와 조인 힌트로 조인 순서와 방법을 가지고 놀자!

- **퀴즈** - 아래의 SQL을 조인 순서는 T1-> T2-> T3->T4로 조인 방법은 NL 조인으로 유도해보자.

```
/*+
Leading((((a b) c) d)) NestLoop(a b) NestLoop(a b c) NestLoop(a b c d)
*/
explain (costs false, analyze)
select a.*, b.*, c.*
from   t1 a, t2 b, t3 c, t4 d
where  a.c1=b.c1 and a.c2=b.c2 and a.c3=b.c3
and    a.c1=c.c1 and a.c2=c.c2 and a.c3=c.c3
and    a.c1=d.c1 and a.c2=d.c2 and a.c3=d.c3;
```

퀴즈 - 힌트를 이용해서 Right Deep Tree를 만들어보자.

```
/*+ Leading((a (b (c d)))) HashJoin(c d) HashJoin(b c d) HashJoin(a b c d) */
explain (costs false, analyze)
select a.*, b.*, c.*
from   t1 a, t2 b, t3 c, t4 d
where  a.c1=b.c1 and a.c2=b.c2 and a.c3=b.c3
and    a.c1=c.c1 and a.c2=c.c2 and a.c3=c.c3
and    a.c1=d.c1 and a.c2=d.c2 and a.c3=d.c3;
      QUERY PLAN
```

실행 순서는 ?

```
-----
Hash Join (actual time=4.127..84.745 rows=100000 loops=1)
-> Seq Scan on t1 a (actual time=0.032..17.478 rows=100000 loops=1)
-> Hash (actual time=4.047..4.047 rows=1000 loops=1)
    Buckets: 1024  Batches: 1  Memory Usage: 278kB
    -> Hash Join (actual time=0.133..3.756 rows=1000 loops=1)
        -> Seq Scan on t2 b
        -> Hash (actual time=0.125..0.125 rows=100 loops=1)
            Buckets: 1024  Batches: 1  Memory Usage: 24kB
            -> Hash Join
                -> Seq Scan on t3 c
                -> Hash
                    Buckets: 1024  Batches: 1
                    -> Seq Scan on t4 d

Planning time: 1.364 ms
Execution time: 96.287 ms
```

퀴즈 - 힌트를 이용해서 Left Deep Tree를 만들어보자.

```
/*+ Leading((((c d) b) a)) HashJoin(c d) HashJoin(c d b) HashJoin(c d b a) */
explain (costs false, analyze)
select a.*, b.*, c.*
from   t1 a, t2 b, t3 c, t4 d
where  a.c1=b.c1 and a.c2=b.c2 and a.c3=b.c3
and    a.c1=c.c1 and a.c2=c.c2 and a.c3=c.c3
and    a.c1=d.c1 and a.c2=d.c2 and a.c3=d.c3;
      QUERY PLAN
```

실행 순서는 ?

```
-----
Hash Join (actual time=1514.549..2703.210 rows=100000 loops=1)
  Hash Cond: ((b.c1 = a.c1) AND (b.c2 = a.c2) AND (b.c3 = a.c3))
  -> Hash Join (actual time=43.870..44.578 rows=1000 loops=1)
    Hash Cond: ((c.c1 = b.c1) AND (c.c2 = b.c2) AND (c.c3 = b.c3))
    -> Hash Join (actual time=0.265..0.419 rows=100 loops=1)
      Hash Cond: ((c.c1 = d.c1) AND (c.c2 = d.c2) AND (c.c3 = d.c3))
      -> Seq Scan on t3 c
      -> Hash (actual time=0.239..0.239 rows=100 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 13kB
        -> Seq Scan on t4 d
    -> Hash (actual time=43.535..43.535 rows=10000 loops=1)
      Buckets: 16384  Batches: 1  Memory Usage: 1545kB
      -> Seq Scan on t2 b
  -> Hash (actual time=1470.484..1470.484 rows=100000 loops=1)
    Buckets: 16384  Batches: 4  Memory Usage: 4023kB
    -> Seq Scan on t1 a (actual time=0.010..25.102 rows=100000 loops=1)
Planning time: 1.161 ms
Execution time: 2713.622 ms
```

퀴즈 - 힌트를 이용해서 Bushy Tree를 만들어보자.

```
/*+ Leading((a c) (b d)) HashJoin(a c) HashJoin(b d) HashJoin(a c b d)*/
explain (costs false, analyze)
select a.*, b.*, c.*
from   t1 a, t2 b, t3 c, t4 d
where  a.c1=b.c1 and a.c2=b.c2 and a.c3=b.c3
and    a.c1=c.c1 and a.c2=c.c2 and a.c3=c.c3
and    a.c1=d.c1 and a.c2=d.c2 and a.c3=d.c3;
      QUERY PLAN
```

실행 순서는 ?

```
-----
Hash Join (actual time=9.333..77.054 rows=100000 loops=1)
  Hash Cond: ((a.c1 = b.c1) AND (a.c2 = b.c2) AND (a.c3 = b.c3))
  -> Hash Join (actual time=0.214..37.697 rows=10000 loops=1)
    Hash Cond: ((a.c1 = c.c1) AND (a.c2 = c.c2) AND (a.c3 = c.c3))
    -> Seq Scan on t1 a (actual time=0.008..15.670 rows=100000 loops=1)
    -> Hash (actual time=0.188..0.188 rows=100 loops=1)
      Buckets: 1024  Batches: 1  Memory Usage: 23kB
      -> Seq Scan on t3 c
  -> Hash (actual time=9.107..9.107 rows=1000 loops=1)
    Buckets: 1024  Batches: 1  Memory Usage: 165kB
    -> Hash Join (actual time=0.409..4.969 rows=1000 loops=1)
      Hash Cond: ((b.c1 = d.c1) AND (b.c2 = d.c2) AND (b.c3 = d.c3))
      -> Seq Scan on t2 b
      -> Hash (actual time=0.397..0.397 rows=100 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 13kB
        -> Seq Scan on t4 d
```

Planning time: 1.117 ms

Execution time: 86.149 ms

BRIN & Partial Index

BRIN 이란?

- BRIN은 PostgreSQL 9.5부터 제공되는 인덱스 유형이다.
- BRIN은 블록 내의 MIN/MAX 값을 이용해서 블록 단위로 인덱싱한다.
- 이로 인해 인덱스 크기는 매우 작아진다는 장점이 있다. 이러한 장점으로 인해, 디스크 공간이 부족한 경우에는 BRIN을 선택하기도 한다.
- BRIN과 EXADATA의 Storage 인덱스는 매우 흡사하다.

블록#1				블록#2				블록#3				블록#4			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

범위#0
최솟값:1 최댓값:8

범위#1
최솟값:9 최댓값:16

범위	최솟값	최댓값
0	1	8
1	9	16

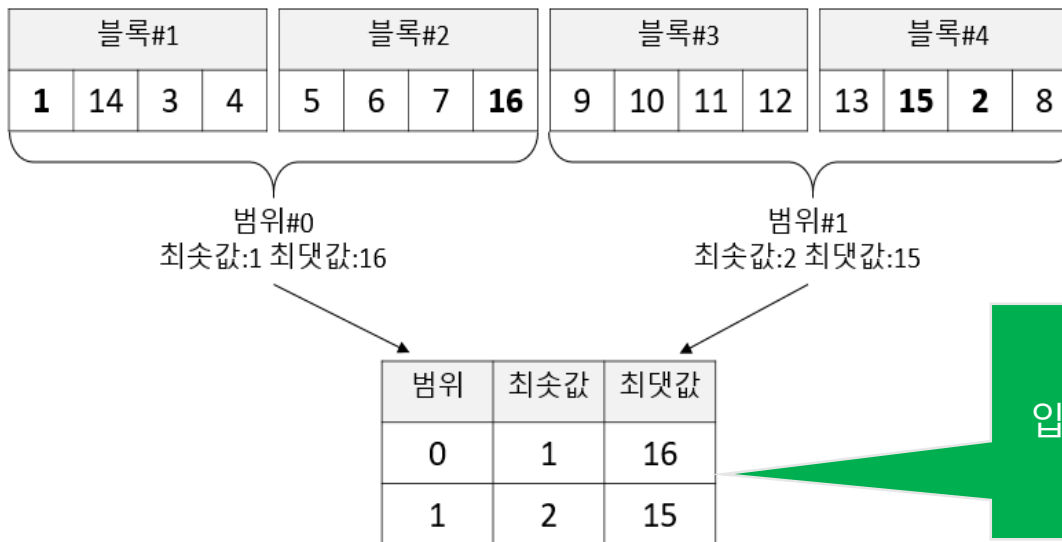
블록 범위 별로 최소, 최댓값을 인덱스에 저장

이때 조건에 5가
입력되면 액세스하는
범위는?

BRIN에서 가장 중요한 것은 정렬이다.

- BRIN의 효과를 극대화하기 위해서는 반드시 정렬이 필요하다.

정렬이 안된 경우



BRIN의 특성

- BRIN의 크기는 매우 작다. 예를 들어, 1Gb 테이블인 경우에 BRIN의 크기는 6블록이다. (1 MB 단위로 1개의 Min/Max 값을 저장하기 때문이다)
- 1건을 읽어도 128 블록 (1 MB)를 읽는다.
- BRIN은 Index Only Scan 방식보다는 느리다. 왜냐면 반드시 테이블을 액세스하기 때문이다.
- BRIN 생성 후에 입력된 레코드들은 Vacuum을 수행해야만 BRIN에 적용된다.
- BRIN은 Row Level 락을 지원한다.

테스트 환경 구성

테스트 환경 구성

```
drop table fact_t;
create table fact_t (
  age integer,           -- 20,30,40,50,60,70 균등입력 (NDV=6)
  gender char(1),        -- M, F 균등입력 (NDV=2)
  blood_type char(2),    -- A,B,O,AB 균등입력 (NDV=4)
  region integer,        -- 1,2,3,4,5,6,7,8 입력 (NDV=8)
  amount integer        -- 랜덤 숫자 입력
);
```

테스트 환경 구성

```
do $$
begin
  for i in 1..130000 loop
    for j in 2..7 loop -- age
      for k in 1..2 loop -- gender
        for l in 1..4 loop -- blood_type
          for m in 1..8 loop -- region
            insert into fact_t
select j*10,
      case when mod(k,2)=0 then 'M'
            when mod(k,2)=1 then 'F'
          end,
      case when mod(l,4)=0 then 'A'
            when mod(l,4)=1 then 'B'
            when mod(l,4)=2 then 'O'
            when mod(l,4)=3 then 'AB'
          end,
      m,
      cast(random()*1000000 as integer);
        end loop;
      end loop;
    end loop;
  end loop;
end$$;
```

테스트 환경 구성

```
create index fact_t_age_brin on fact_t using brin (age);
create index fact_t_gender_brin on fact_t using brin (gender);
create index fact_t_blood_brin on fact_t using brin (blood_type);
create index fact_t_region_brin on fact_t using brin (region);
analyze fact_t;
```

```
select relname, relpages, round(relpages*8/1024,0) "size(MiB)"
from   pg_class where relname like 'fact_t%';
```

relname	relpages	size(MiB)
fact_t	317962	2484
fact_t_age_brin	10	0
fact_t_blood_brin	10	0
fact_t_gender_brin	10	0
fact_t_region_brin	10	0

BRIN 크기는
10 블록

정렬 전 Explain 결과

```
explain (costs false, analyze, buffers)
select age, gender, blood_type, region, count(*), sum(amount)
from   fact_t
where  age=30
and    gender='F'
and    blood_type='AB'
and    region=2
group by age, gender, blood_type, region;
```


정렬 전 Explain 결과

QUERY PLAN

```
-----
GroupAggregate (actual time=87521.104..87521.105 rows=1 loops=1)
  Group Key: age, gender, blood_type, region
  Buffers: shared hit=20 read=317973 written=3
  -> Bitmap Heap Scan on fact_t (actual time=60.441..87351.753 rows=130000
loops=1)
    Recheck Cond: ((region = 2) AND (age = 30))
    Rows Removed by Index Recheck: 48880000
    Filter: ((gender = 'F'::bpchar) AND (blood_type = 'AB'::bpchar))
    Rows Removed by Filter: 910000
    Heap Blocks: lossy=317962
    Buffers: shared hit=20 read=317973 written=3
    -> BitmapAnd (actual time=59.416..59.416 rows=0 loops=1)
      Buffers: shared hit=20 read=11
      -> Bitmap Index Scan on fact_t_region_brin (actual
time=25.965..25.965 rows=3180800 loops=1)
        Index Cond: (region = 2)
        Buffers: shared hit=16 read=4
      -> Bitmap Index Scan on fact_t_age_brin (actual
time=26.823..26.823 rows=3180800 loops=1)
        Index Cond: (age = 30)
        Buffers: shared hit=4 read=7
    Planning time: 5.345 ms
    Execution time: 87524.162 ms
```

수행 시간은 87.5초

데이터 정렬

```
create table fact_t2
as select * from fact_t order by age, gender, blood_type, region;

create index fact_t2_age_brin      on fact_t2 using brin (age);
create index fact_t2_gender_brin  on fact_t2 using brin (gender);
create index fact_t2_blood_brin   on fact_t2 using brin (blood_type);
create index fact_t2_region_brin  on fact_t2 using brin (region);
```

데이터 정렬 후에는 BITAND 연산을 통해 극적으로 빨라진다.

```
explain (costs false, analyze, buffers)
select age, gender, blood_type, region, count(*), sum(amount)
from   fact_t2
where  age=30
and    gender='F'
and    blood_type='O'
and    region=3
group by age, gender, blood_type, region;
```

데이터 정렬 후에는 BITAND 연산을 통해 극적으로 빨라진다.

QUERY PLAN

GroupAggregate (actual time=283.639..283.640 rows=1 loops=1)

Group Key: age, gender, blood_type, region

Buffers: shared hit=27 read=7582

-> **Bitmap Heap Scan on fact_t2**

Recheck Cond: ((region = 3) AND (blood_type = 'O'::bpchar) AND (gender = 'F'::bpchar))

Rows Removed by Index Recheck: 405664

Filter: (age = 30)

Rows Removed by Filter: 650000

Heap Blocks: lossy=7552

Buffers: shared hit=27 read=7582

-> **BitmapAnd** (actual time=28.167..28.167 rows=0 loops=1)

Buffers: shared hit=27 read=30

-> **Bitmap Index Scan on fact_t2_region_brin** (rows=518400 loops=1)

Index Cond: (region = 3)

Buffers: shared hit=12 read=11

-> **Bitmap Index Scan on fact_t2_blood_brin** (rows=814080 loops=1)

Index Cond: (blood_type = 'O'::bpchar)

Buffers: shared hit=13 read=10

-> **Bitmap Index Scan on fact_t2_gender_brin** (rows=1597440 loops=1)

Index Cond: (gender = 'F'::bpchar)

Buffers: shared hit=2 read=9

Planning time: 9.584 ms

Execution time: 284.492 ms

수행 시간은 0.3초

Partial Index

- PostgreSQL은 인덱스 생성 시에 WHERE 조건을 제공한다.
- PostgreSQL은 NULL 값도 인덱스에 포함된다.

```
drop table t10;
create table t10 (c1 integer, flag char(1), dummy char(10));
insert into t10 select i, 'Y', 'dummy' from generate_series(1,10000000) a(i);
insert into t10 select i, 'N', 'dummy' from generate_series(100000001,100)
a(i);
```

```
create index t10_flag_idx on t10(flag) where flag='N' ;
```

```
explain select * from t10 where flag='Y';
          QUERY PLAN
```

```
-----
Seq Scan on t10 (cost=0.00..188695.00 rows=10000000 width=17)
  Filter: (flag = 'Y'::bpchar)
```

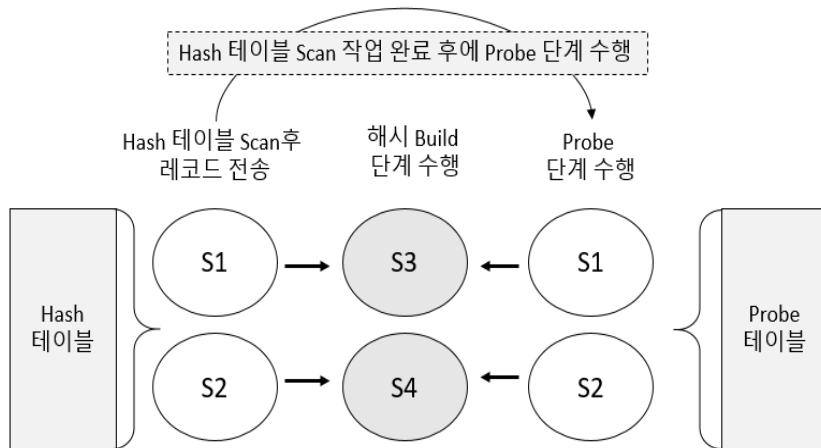
```
postgres=# explain select * from t10 where flag='N';
          QUERY PLAN
```

```
-----
Index Scan using t10_flag_idx on t10 (cost=0.12..4.14 rows=1 width=17)
```

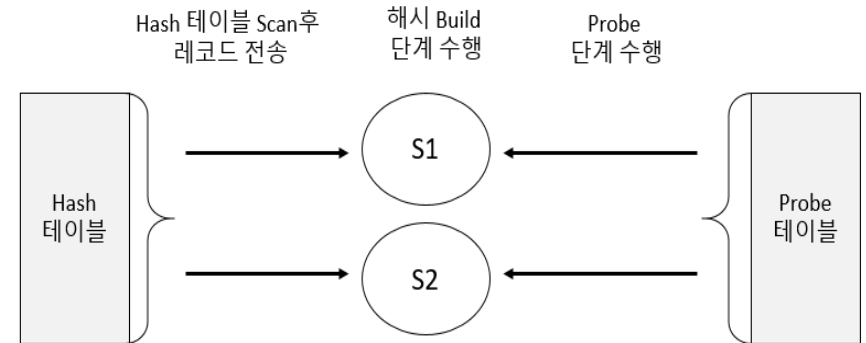
Parallel Processing

Parallel Processing 개요

- PostgreSQL 9.6 부터 병렬 처리를 지원한다.
- 병렬 처리는 Parallel Scan, Parallel Group by, Parallel Join을 지원한다.
- ORACLE과 같은 생산자-소비자 모델은 아니다.



ORACLE 방식



PostgreSQL 방식

Parallel 처리 관련 파라미터

파라미터	설명
max_worker_processes	Parallel 처리를 수행하는 worker 프로세스의 최대 개수를 설정한다.
max_parallel_workers_per_gather	하나의 쿼리 당 수행 가능한 최대 worker 프로세스 개수를 설정한다. 기본 설정값은 0이므로 어떤 경우에도 Parallel 처리를 하지 않는다.
force_parallel_mode	max_parallel_workers_per_gather 파라미터값이 0보다 크면, 옵티마이저는 쿼리 비용을 계산한 후에 Parallel 처리 여부를 결정한다. 만일 이 값을 on으로 설정하면 항상 Parallel 처리를 고려한다. 기본 설정값은 off이다.
parallel_setup_cost	Parallel 처리를 하려면 worker 프로세스를 할당받기 위한 사전 작업이 필요하다. 이 파라미터는 이러한 사전 작업에 필요한 비용을 의미한다. 기본 설정값은 1000이다.

Parallel 처리 관련 파라미터

파라미터	설명
parallel_tuple_cost	Parallel 처리를 위해서는 부모 프로세스와 worker 프로세스 간에 레코드 전송이 필요하다. 이때, 전송되는 레코드 개수가 많다면 Parallel 처리 시에 부담이 될 수 있다. 이 파라미터의 목적은 처리해야 할 레코드 수가 많을 때, 싱글 프로세스로 처리하도록 유도하는 것이다. 기본 설정값은 0.1이다. max_parallel_workers_per_gather 파라미터가 0보다 클 때, 이 값을 0으로 설정하면 항상 Parallel 방식으로 처리한다.
min_parallel_relation_size	Parallel 처리를 위한 최소 테이블 크기를 설정한다. 기본 설정값은 8 MiB이다. 즉, 8 MiB 이상인 테이블은 Parallel 처리 대상이다.

Worker 프로세스 개수 산정 방식

- Worker 프로세스의 개수는 min_parallel_relation_size 파라미터값을 기준으로 산정한다.
- 기준 크기의 3배 단위로 1개씩 증가하며, 최대 7개이다.

테이블 크기 (MiB)	Worker 개수
< 8	0
< 24	1
< 72	2
< 216	3
< 648	4
< 1944	5
< 5822	6
> = 5822	7

Parallel Scan 예제

```
set max_parallel_workers_per_gather to 8;  
explain (costs false, analyze, buffers)  
select count(*) from t2;
```

QUERY PLAN

```
-----  
Finalize Aggregate (actual time=33920.893..33920.893 rows=1 loops=1)  
  Buffers: shared hit=567 read=1428572  
    -> Gather (actual time=33919.719..33920.887 rows=8 loops=1)  
      Workers Planned: 7  
      Workers Launched: 7  
      Buffers: shared hit=567 read=1428572  
        -> Partial Aggregate  
          (actual time=33910.839..33910.839 rows=1 loops=8)  
            Buffers: shared read=1428572  
              -> Parallel Seq Scan on t2  
                (actual time=0.441..33690.431 rows=1250000 loops=8)  
                  Buffers: shared read=1428572  
  
Planning time: 0.040 ms  
Execution time: 33921.497 ms
```

PostgreSQL 10 - Parallel Index Scan

- PostgreSQL 10부터는 Parallel Index Scan 기능을 제공한다.
- Parallel Index Scan 기능은 Index Full Scan 뿐만 아니라 Index Range Scan 시에도 동작한다.
- 처리 알고리즘은 다음과 같다.

- 1) 조건절 범위에 해당하는 "Start 리프 블록"과 "End 리프 블록"의 위치를 계산한다.
- 2) 처리 범위가 인덱스 병렬 처리를 할 정도로 큰지 확인한다. (아래 표 참조)
- 3) 만일 크다면, 크기에 따라서 Worker 개수를 설정한 후에,
"Start" ~ "End" 범위를 나눠서 처리한다.
- 4) 만일 작다면, 싱글 프로세스로 처리한다.

표-1. Parallel Worker 개수

테이블 블록 수		Worker 개수
<	1,024	0
<	3,072	1
<	9,216	2
<	27,648	3
<	82,944	4
<	248,832	5
<	746,496	6
>=	746,496	7

인덱스 처리 블록수		Worker 개수
<	64	0
<	192	1
<	576	2
<	1,728	3
<	5,184	4
<	15,552	5
<	46,656	6
>=	46,656	7

PostgreSQL 10 - Parallel Index Scan 예제

```
[DEBUG_INDEX_PQ] heap_pages=[23249.000000] min_parallel_table_scan_size=[1024]
index_pages=[2703.000000] min_parallel_index_scan_size=[64]
```

```
set max_parallel_workers_per_gather to 8;
```

```
explain (analyze, buffers)
select count(*) from np1 where c1 between 1000000 and 2000000 and
dummy='dummy';
QUERY PLAN
```

```
-----
Finalize Aggregate  (cost=32710.82..32710.83 rows=1 width=0)
  Buffers: shared hit=12035
  -> Gather  (cost=32710.50..32710.81 rows=3 width=8)
    Workers Planned: 3
    Workers Launched: 3
    Buffers: shared hit=12035
    -> Partial Aggregate
      Buffers: shared hit=11612
      -> Parallel Index Scan using np1_c1_idx on np1
        Index Cond: ((c1 >= 1000000) AND (c1 <= 2000000))
        Filter: (dummy = 'dummy'::bpchar)
        Buffers: shared hit=11612
```

테이블 크기와 인덱스
범위에 해당되는 PQ
개수를 각각 계산한 후에
더 적은 수로 설정한다.

```
Planning time: 0.123 ms
```

```
Execution time: 264.310 ms
```

PostgreSQL 10 - Parallel Index Scan 예제

```
set max_parallel_workers_per_gather=0;
```

```
explain (analyze, buffers)
select count(*) from np1 where c1 between 1000000 and 2000000 and
dummy='dummy' ;
```

QUERY PLAN

```
-----
Aggregate  (cost=41721.38..41721.39 rows=1 width=8)
  Buffers: shared hit=9105
    ->  Index Scan using np1_c1_idx on np1
          Index Cond: ((c1 >= 1000000) AND (c1 <= 2000000))
          Filter: (dummy = 'dummy'::bpchar)
          Buffers: shared hit=9105
Planning time: 0.119 ms
Execution time: 431.169 ms
```

싱글 프로세스로
처리하면 Parallel 처리에
비해서 80% 정도
느려진다.

토의