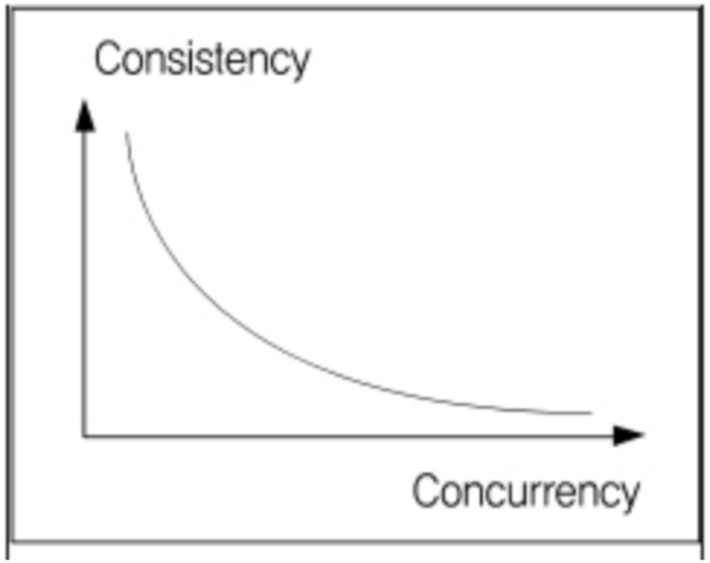
Postgresql 의 Mvcc와 조회수 로직 개선 방향

동시성 제어란?

동시성 제어란 `DBMS가 다수의 사용자 사이에서 동시에 작용하는 다중 트랜잭션의 상호간섭 작용에서 Database를 보호하는 것`을 의미한다.



일반적으로 동시성과 일관성은 반비례 관계를 가지고 있다.

동시성 제어 분류

낙관적 동시성 제어

- 사용자들이 동시에 같은 데이터를 수정하는 경우가 적을 것으로 가정
- 락을 걸지 않고, 수정 반영 시점에 값이 변경되었는지 검사

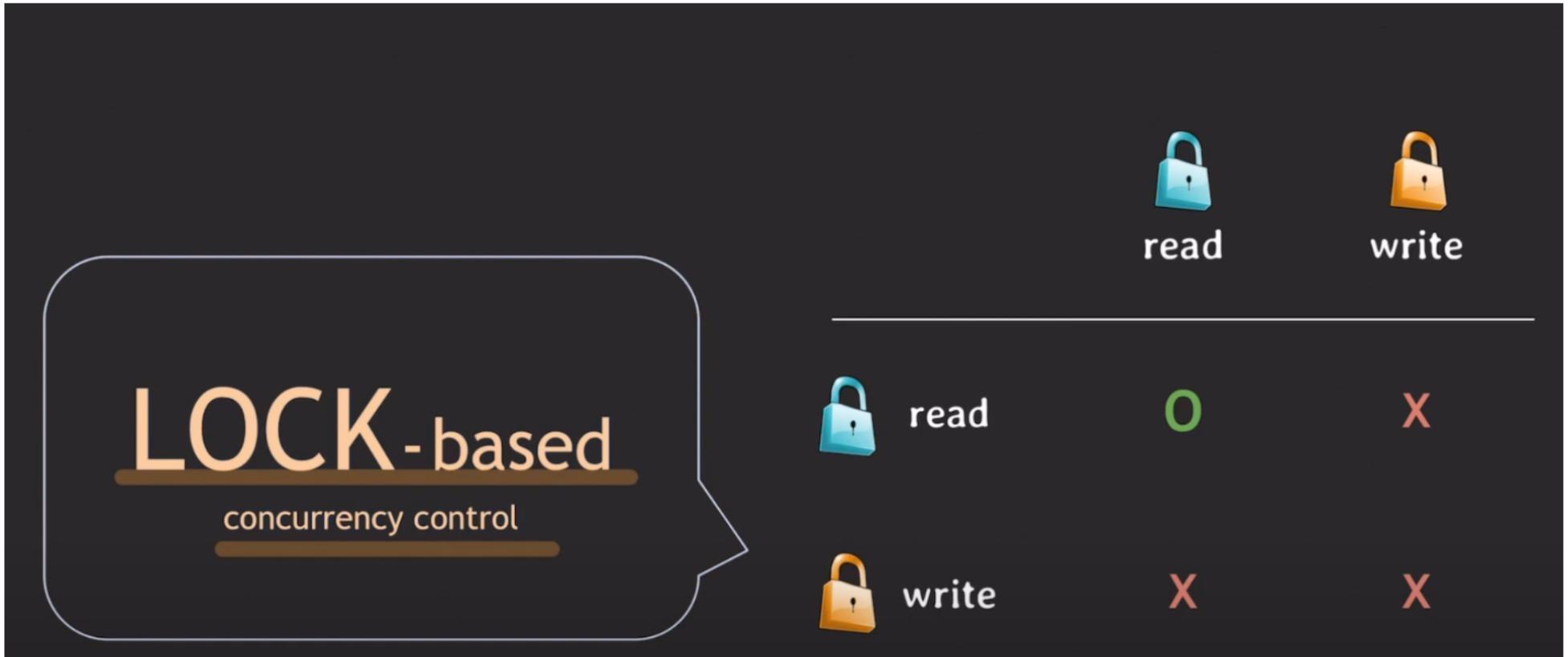
비관적 동시성 제어

- 동시에 같은 데이터를 수정할 것이라고 가정
- Lock을 걸고 동시수정을 명시적으로 막음
- 동시성이 비교적 떨어짐

공유락과 배타락

비관적 동시성 제어를 위한 대표적인 방법으로 Lock이 있는데, 여기에는 두가지 종류가 있다

- 공유락:Shared Lock:읽기 잠금
- 배타락:Exclusive Lock:쓰기 잠금



락 기반 동시성 제어의 문제점

- 읽기와 쓰기 작업이 서로 방해를 일으키기 때문에, 동시성 문제가 발생한다.
- 동시성 저하가 크게 발생한다.

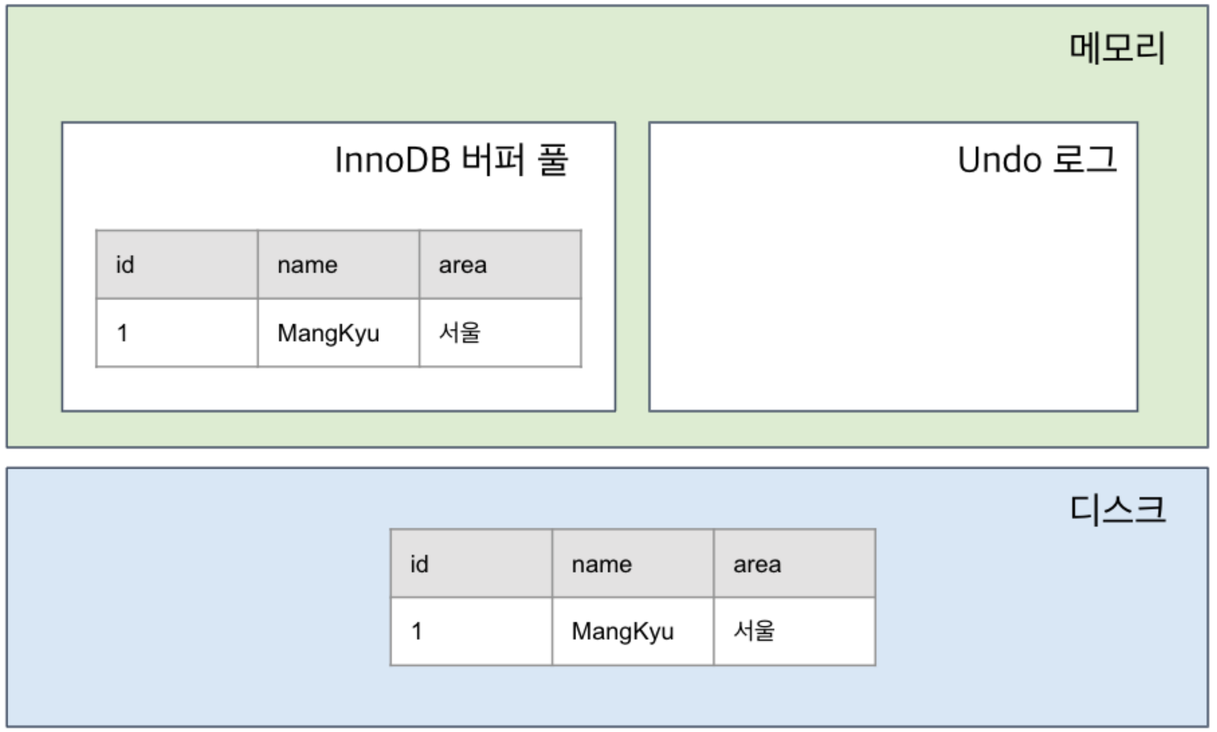
이러한 문제점을 해결하기 위해 MVCC(Multi-Version Concurrency Control) 이 탄생하게 되었다.

MVCC(Multi-Version Concurrency Control)

대부분의 DBMS에서 동시성 수행을 위해 제공하는 MVCC 기능은

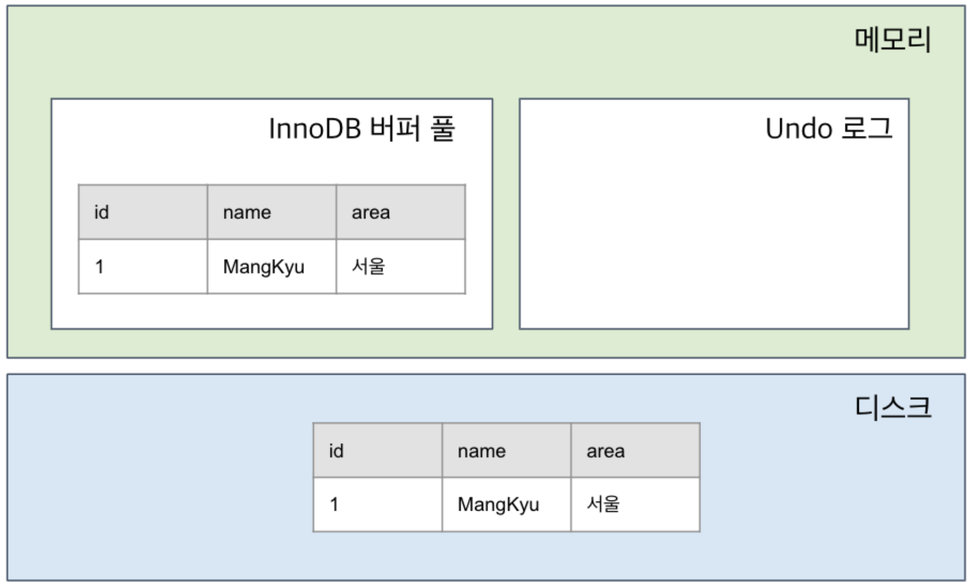
- 동시에 여러 트랜잭션이 수행되는 환경에서,
- 각 트랜잭션에게 쿼리 수행시점의 데이터를 보장해줌과 동시에,
- Read/Write 간의 충돌 및 Lock을 방지하여 동시성을 높일 수 있는 기능이다.

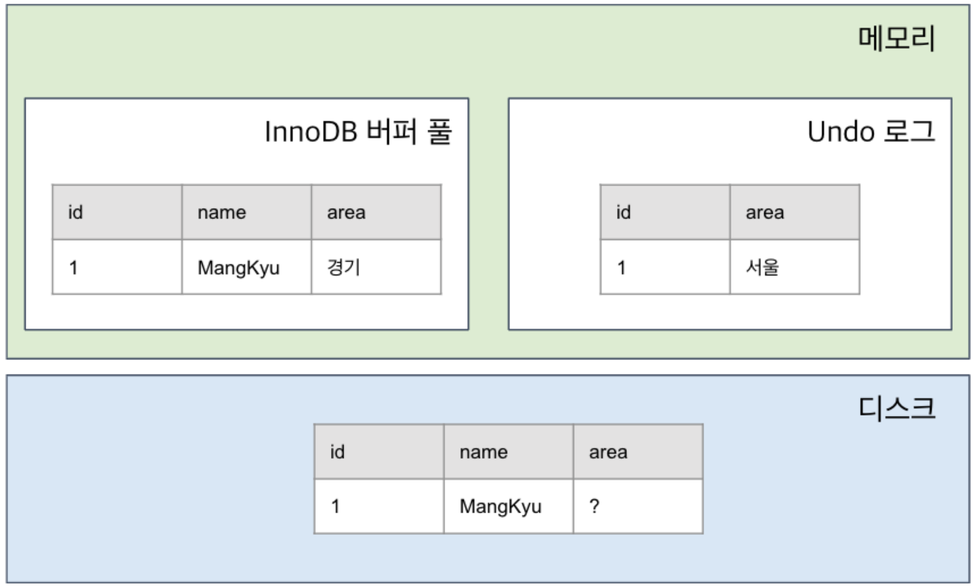
MySql의 Mvcc 는 어떻게 동작할까?




```
UPDATE member SET area = "경기" WHERE id = 1;
```

MySql의 Mvcc 는 어떻게 동작할까?

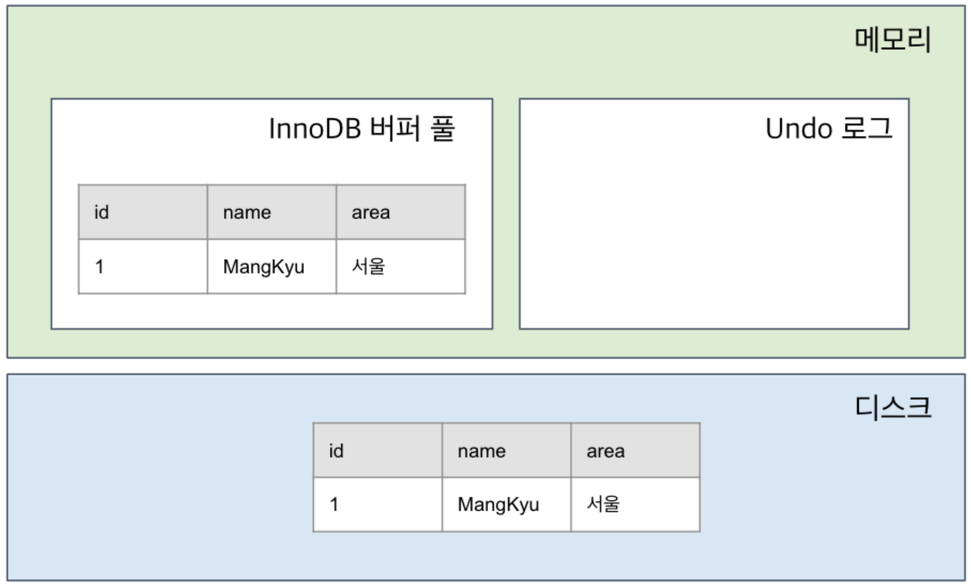


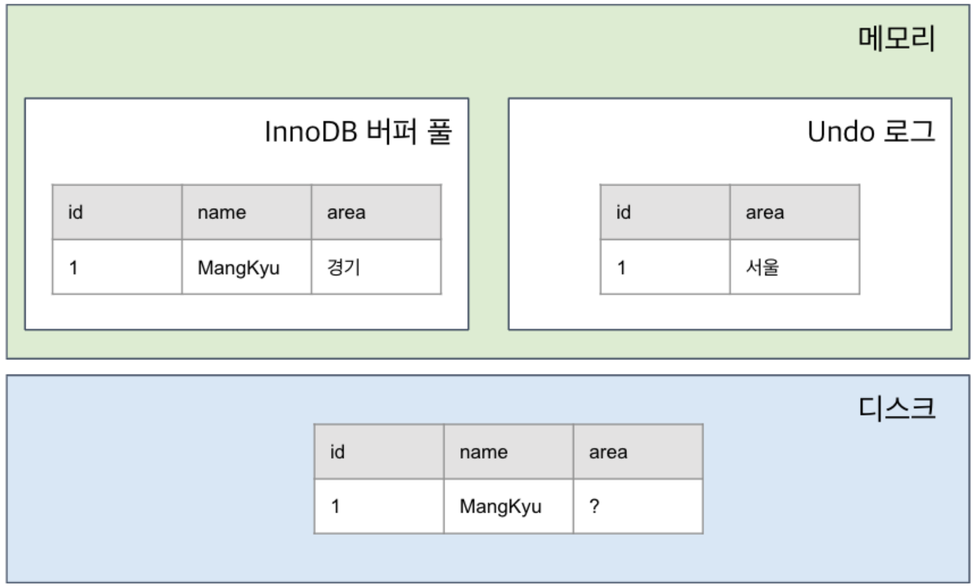



```sql
UPDATE member SET area = "경기" WHERE id = 1;
```

MySql의 Mvcc 는 어떻게 동작할까?

이 상태에서 SELECT 문이 나온다면 뭐가 나올까?






```
UPDATE member SET area = "경기" WHERE id = 1;
```

오늘의 핵심 주제 : Postgresql 에서 MVCC는?

오늘의 핵심 주제 : Postgresql 에서 MVCC는?

```
 xmin  | xmax  |  value
-------+-------+-----
  2010 |  2020 | AAA
  2012 |     0 | BBB
  2014 |  2030 | CCC
  2020 |     0 | ZZZ
```

이 상태에서 transaction_id가 2016인 사람이 SELECT를 한다면?

Dead Tuple


Tuple 1
Tuple 2
...
value=1
xmin=100
xmax=null
value=2
xmin=101
xmax=null
update
UPDATE tb_test
SET value = 2_update
WHERE value = 2;
Tuple 1
Tuple 2(Old)
Tuple 2`(new)
...
value=1
xmin=100
xmax=null
value=2
xmin=101
xmax=102
value=
2_update
xmin=102
xmax=null

실제로 한번 봅시다.

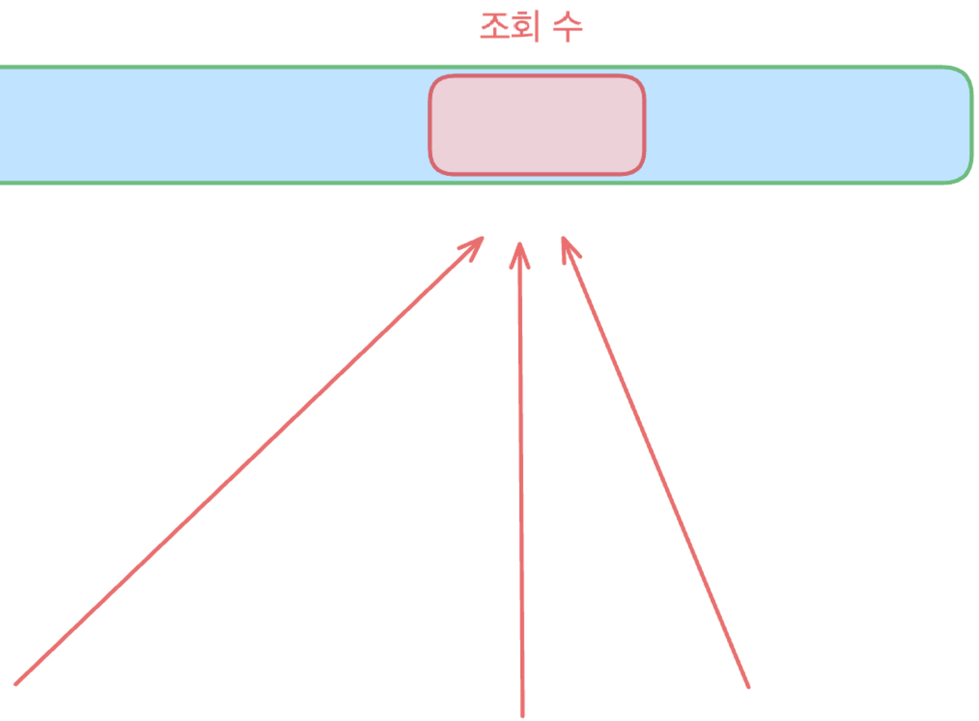
```
### autovacuum off
testdb=> alter table tb_test set (autovacuum_enabled = off);
ALTER TABLE

### delete 전 count 조회
testdb=> select count(*) from tb_test;
  count
----------
 10000000
(1 row)
Time: 548.779 ms

### delete 후 count 조회
testdb=> delete from tb_test where ai != 1;
DELETE 9999999
Time: 30142.916 ms (00:30.143)
```

```
testdb=> select count(*) from tb_test;
 count
-------
     1
(1 row)
Time: 497.835 ms
=> 1건을 count하는데도 오래걸림

### Dead Tuple 확인
testdb=> SELECT relname, n_live_tup, n_Dead_tup, n_Dead_tup / (n_live_tup::float) as ratio
FROM pg_stat_user_tables;
 relname  | n_live_tup | n_Dead_tup |  ratio
----------+------------+------------+----------
 tb_test  |          1 |    9999999 |  9999999
=> Dead Tuple이 9999999개로 증가함, 반면에 live Tuple은 1건

### 테이블 사이즈 조회
testdb=> dt+ tb_test
                              List of relations
 Schema |  Name   | Type  | Owner  | Persistence |  Size   | Description
--------+---------+-------+--------+-------------+---------+-------------
 public | tb_test | table | master | permanent   | 1912 MB |
=> 1건짜리 테이블임에도 거의 2GB인 상황
```

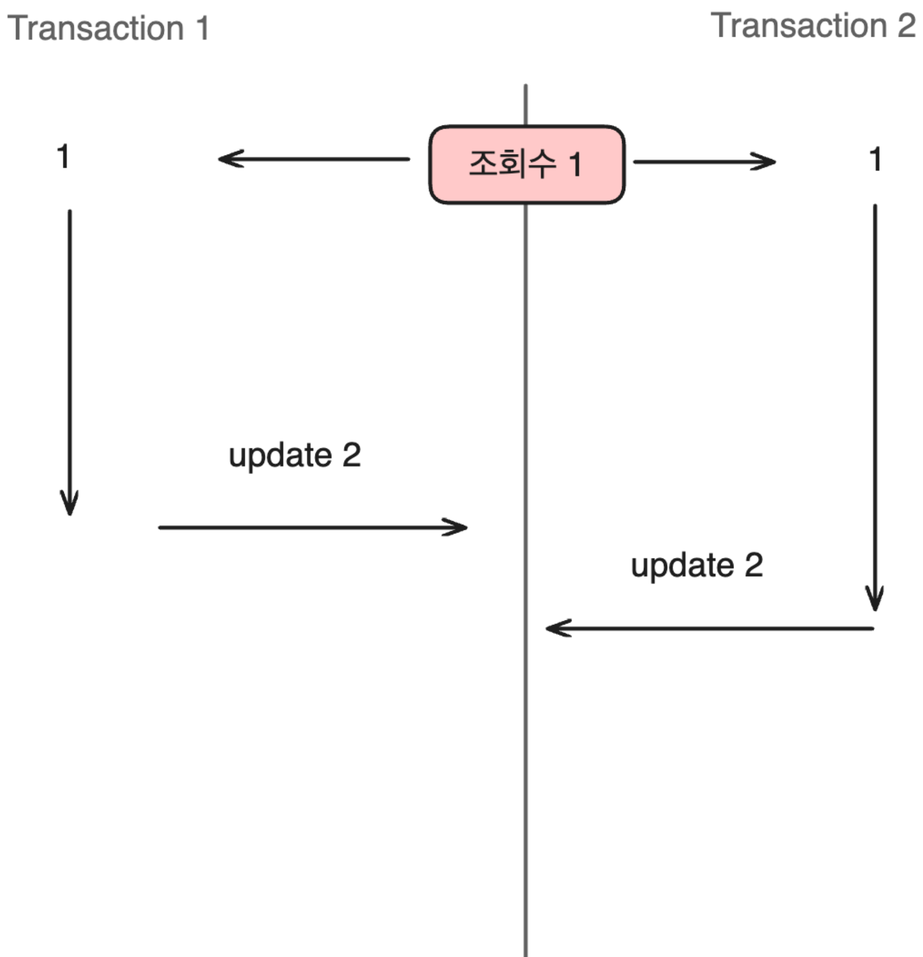
실제로 한번 봅시다.

```
### vacuum 수행 후 Dead Tuple 확인
testdb=> vacuum tb_test;
VACUUM
Time: 11273.959 ms (00:11.274)

testdb=> SELECT relname, n_live_tup, n_Dead_tup, n_Dead_tup / (n_live_tup::float) as ratio
FROM pg_stat_user_tables;
 relname  | n_live_tup | n_Dead_tup | ratio
----------+------------+------------+-------
 tb_test  |          1 |          0 |     0
=> Dead Tuple이 0개로 모두 정리됨

### 데이터 조회 속도 및 테이블 사이즈 비교
testdb=> select count(*) from tb_test;
 count
-------
     1
(1 row)
Time: 9.971 ms
=> 앞에서와 달리 1건 counting이 바로 완료됨

testdb=> dt+ tb_test;
                            List of relations
 Schema |  Name   | Type  | Owner  | Persistence | Size  | Description
--------+---------+-------+--------+-------------+-------+-------------
 public | tb_test | table | master | permanent   | 1912 MB |
=> vacuum 수행 후 dead tuple은 정리되었지만 테이블 사이즈는 그대로임

### vacuum full 수행 후 테이블 사이즈 확인
testdb=> vacuum full tb_test;
VACUUM

testdb=> dt+ tb_test;
                            List of relations
 Schema |  Name   | Type  | Owner  | Persistence | Size  | Description
--------+---------+-------+--------+-------------+-------+-------------
 public | tb_test | table | master | permanent   | 40 kB |
=> vacuum full 수행 후에는 테이블 사이즈도 줄어들었음
```

조회수 로직이라면 어떻게 적용이 될까?

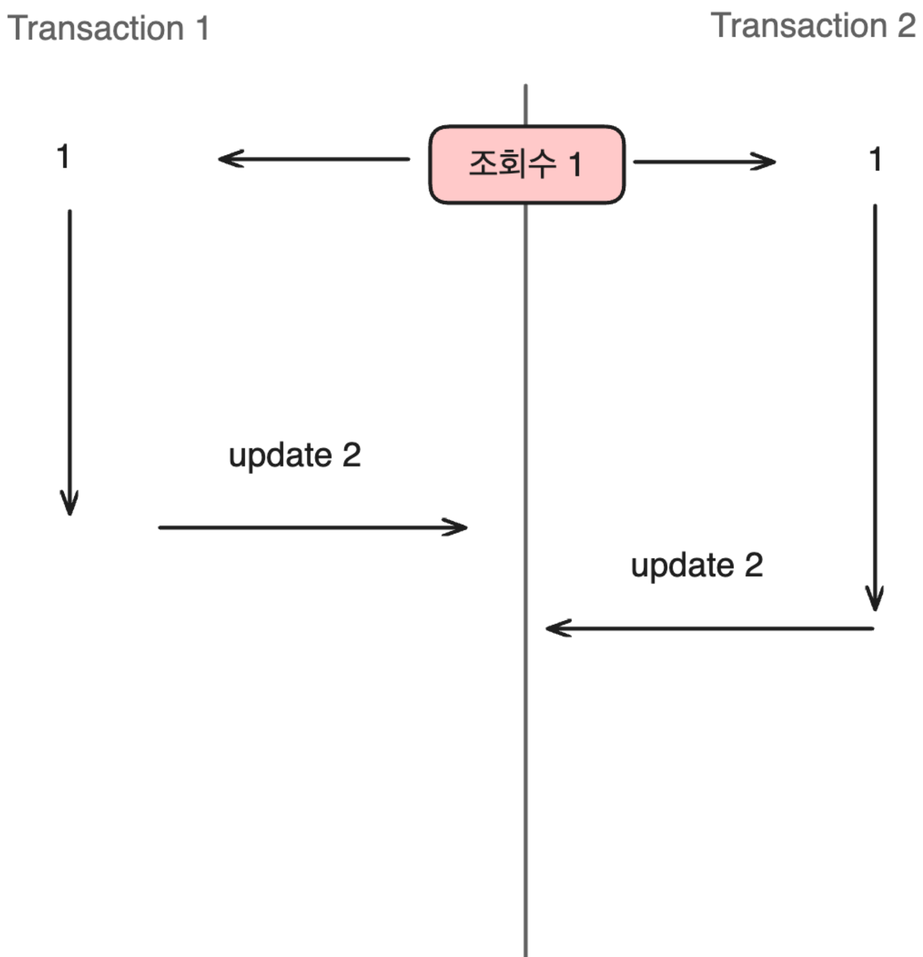


조회수 로직이라면 어떻게 적용이 될까?



락이든 mvcc 든 도입을 하게 된다.

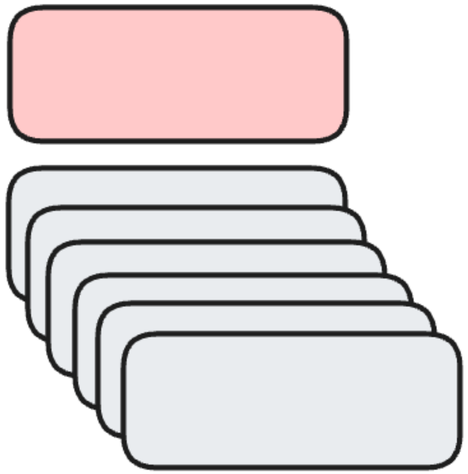
조회수 로직이라면 어떻게 적용이 될까?



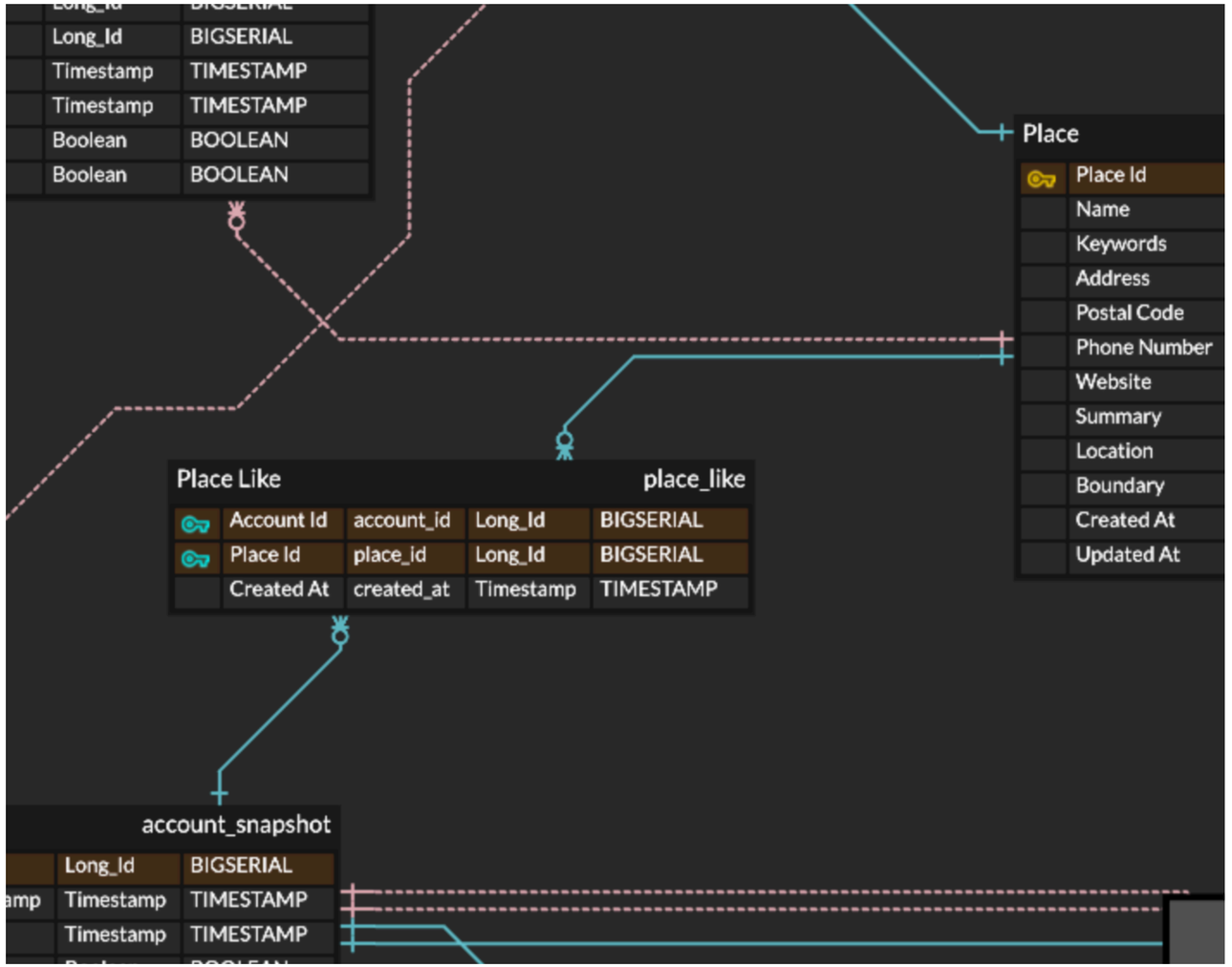
락이든 mvcc 든 도입을 하게 된다.

Postgresql Mvcc에서

dead tuple



매핑 테이블 방식


Long_Id BIGSERIAL
Timestamp TIMESTAMP
Timestamp TIMESTAMP
Boolean BOOLEAN
Boolean BOOLEAN
Place
Place Id
Name
Keywords
Address
Postal Code
Phone Number
Website
Summary
Location
Boundary
Created At
Updated At
Place Like
place_like
Account Id account_id Long_Id BIGSERIAL
Place Id place_id Long_Id BIGSERIAL
Created At created_at Timestamp TIMESTAMP
account_snapshot
Long_Id BIGSERIAL
Timestamp TIMESTAMP
Timestamp TIMESTAMP

Redis Hyper log log

하이퍼로그로그

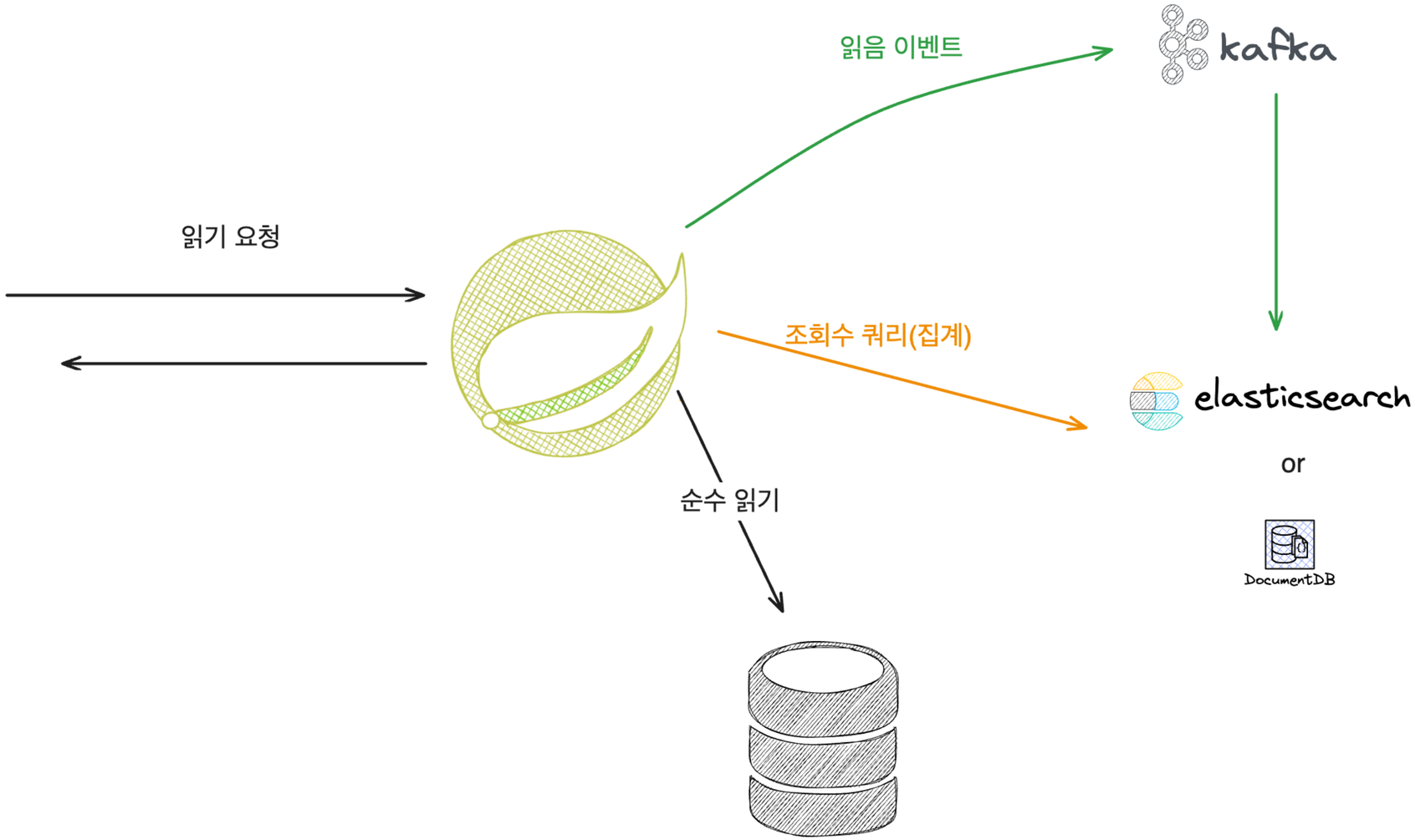
HyperLogLog는 세트의 카디널리티를 추정하는 확률적 데이터 구조입니다.

HyperLogLog는 세트의 카디널리티를 추정하는 확률적 데이터 구조입니다. 확률론적 데이터 구조인 HyperLogLog는 효율적인 공간 활용을 위해 완벽한 정확성을 제공합니다.

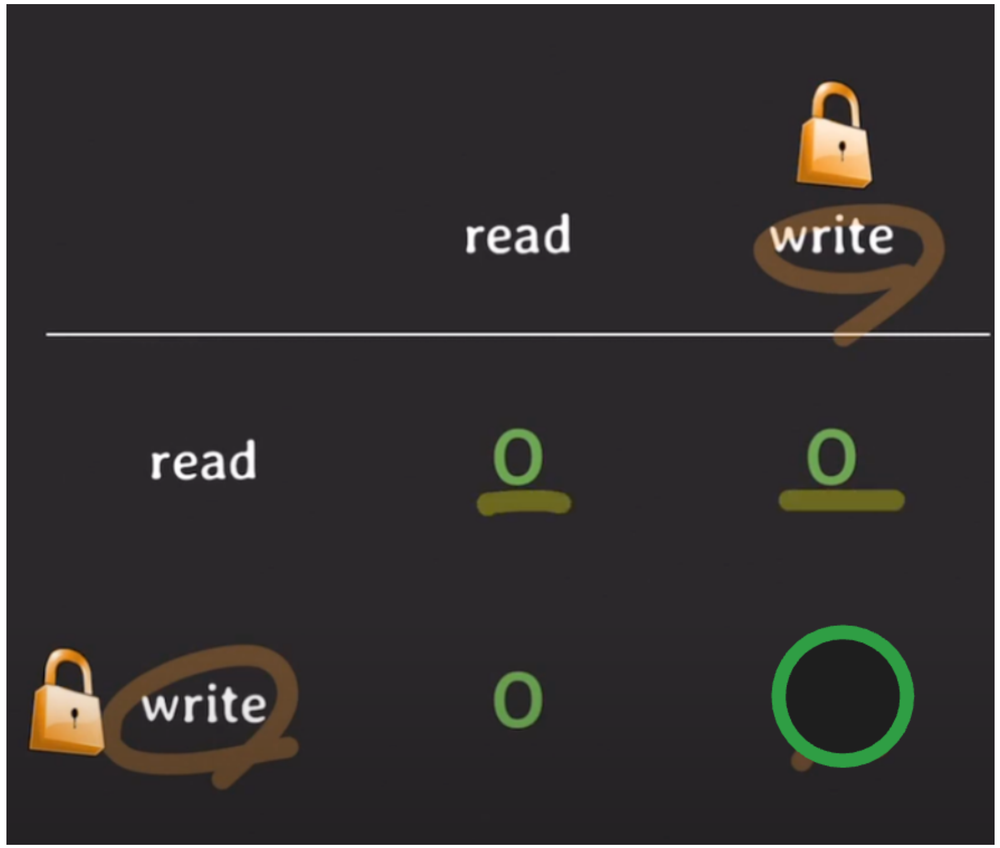
Redis HyperLogLog 구현은 최대 12KB를 사용하고 0.81%의 표준 오류를 제공합니다.

고유 항목을 계산하려면 일반적으로 계산하려는 항목 수에 비례하는 메모리 양이 필요합니다. 여러 번 계산하지 않으려면 과거에 이미 본 요소를 기억해야 하기 때문입니다. 그러나 정밀도를 위해 메모리를 교환하는 일련의 알고리즘이 존재합니다. 이는 표준 오류가 있는 추정 측정값을 반환하며, HyperLogLog에 대한 Redis 구현의 경우 1% 미만입니다. 이 알고리즘의 마법은 더 이상 계산된 항목 수에 비례하는 메모리 양을 사용할 필요가 없고 대신 일정한 양의 메모리를 사용할 수 있다는 것입니다. 최악의 경우 12k 바이트이거나 HyperLogLog(지금부터 HLL이라고 부르겠습니다)에 요소가 거의 없는 경우 훨씬 적습니다.

어떤 방식으로 개선하려고?



그러면 어떻게 달라지는가



updatable
디테일을 잃는다

64

update 작업 절대 없음
디테일을 보관하고 있다 -> 추후 요구사항 유연하게 수용가능

- 언제, 누가, 어떤 글을 읽었는지
- 언제, 누가, 어떤 글을 읽었는지
- 언제, 누가, 어떤 글을 읽었는지
- 언제, 누가, 어떤 글을 읽었는지

끝