데이터베이스시스템
Term Project

2019320097 - 조이강

Lab setup

- You can create a partitioned table with a query like this
(But be careful. The partition key must be contained inside the primary key!)
  - CREAT TABLE [tablename] {…
…} PARTITION BY RANGE [attributename]; // for rangable data

    CREAT TABLE [tablename]_[partitionname] PARTITION OF [tablename]
FOR VALUES FROM ([from]) TO ([to]);

    CREATE TABLE [tablename] { …
… } PARTITION BY LIST [attributename]; // for listable data

    CREAT TABLE [tablename]_[partitionname] PARTITION OF [tablename]
FOR VALUES IN ([attributename]);

- SET enable_bitmapscan=false;
SET max_parallel_workers_per_gather=0;
- Now, Make two log tables by using following codes.

CREATE TABLE Log_unpart (id SERIAL,
date DATE NOT NULL,
type VARCHAR(10) NOT NULL CHECK (type IN ('send',
- 'receive')), amount NUMERIC NOT NULL,
PRIMARY KEY (date, id));

- CREATE TABLE Log_part (
id SERIAL,
date DATE NOT NULL,
type VARCHAR(10) NOT NULL CHECK (type IN ('send', 'receive')),
amount NUMERIC NOT NULL,
PRIMARY KEY (date, id)
) PARTITION BY RANGE (date);

```
d2019320097=# create table Log_unpart (
d2019320097(# id serial,
d2019320097(# date DATE NOT NULL,
d2019320097(# type VARCHAR(10) NOT NULL CHECK (type IN('send', 'receive')),
d2019320097(# amount Numeric not null,
d2019320097(# PRIMARY KEY (date, id)
d2019320097(# );
d2019320097=# CREATE TABLE Log_part (
d2019320097(#     id SERIAL,
d2019320097(#     date DATE NOT NULL,
d2019320097(#     type VARCHAR(10) NOT NULL CHECK (type IN ('send', 'receive')),
d2019320097(#     amount NUMERIC NOT NULL,
d2019320097(#     PRIMARY KEY (date, id) -- 파티션 키와 기본 키를 함께 포함
d2019320097(# ) PARTITION BY RANGE (date);
d2019320097=# CREATE TABLE Log_part_q1 PARTITION OF Log_part FOR VALUES FROM ('2024-01-01') TO ('2024-04-01');
CREATE TABLE
d2019320097=# CREATE TABLE Log_part_q2 PARTITION OF Log_part FOR VALUES FROM ('2024-04-01') TO ('2024-07-01');
CREATE TABLE
d2019320097=# CREATE TABLE Log_part_q3 PARTITION OF Log_part FOR VALUES FROM ('2024-07-01') TO ('2024-10-01');
CREATE TABLE
d2019320097=# CREATE TABLE Log_part_q4 PARTITION OF Log_part FOR VALUES FROM ('2024-10-01') TO ('2025-01-01');
CREATE TABLE
d2019320097=#
```

- CREATE TABLE Log_part_q1 PARTITION OF Log_part FOR VALUES FROM ('2024-01-01') TO ('2024-04-01');
- CREATE TABLE Log_part_q2 PARTITION OF Log_part FOR VALUES FROM ('2024-04-01') TO ('2024-07-01');
- CREATE TABLE Log_part_q3 PARTITION OF Log_part FOR VALUES FROM ('2024-07-01') TO ('2024-10-01');
- CREATE TABLE Log_part_q4 PARTITION OF Log_part FOR VALUES FROM ('2024-10-01') TO ('2025-01-01');

Table Information

- Log Table's schema is as follows.

Attribute	Data Type	Constraints
ID	SERIAL	PRIMARY KEY
Date	DATE	NOT NULL, PRIMARY KEY
Type	VARCHAR(10)	NOT NULL, CHECK(type IN ('send', 'receive'))
Amount	NUMERIC	NOT NULL

Table name	Range
Log_part_q1	2024-01-01 ~ 2024-04-01
Log_part_q2	2024-04-01 ~ 2024-07-01
Log_part_q3	2024-07-01 ~ 2024-10-01
Log_part_q4	2024-10-01 ~ 2025-01-01

- But, log_part table is separated into four partitions.
  Each partition, log_part_q1, log_part_q2, log_part_q3, and log_part_q4,
  is separated by date in the log table, with each table having data for three months.

EXERCISE 1

I. Use the following code to add 100,000 random records to log_unpart and log_part.

```
INSERT INTO [tablename] (date, type, amount)
SELECT ('2024-01-01'::DATE + (random() * 364)::INT) AS date,
CASE WHEN random() < 0.5 THEN 'send' ELSE 'receive' END AS type, round(random() * 1000) AS amount
 FROM generate_series(1, 100000);
```

1) Insert random records in log_unpart
2) Insert random records in log_part

II. Verify that the created records inserted into the partition that matches the range.

1) Make query that include a specific quarter.

   Ex) select all record from '2024-04-01' to '2024-06-30'

2) Using the same query in above, expect the execution time and plan and use EXPLAIN ANLAYZE to analyze them.

III. Make a range query that includes multiple quarters.

1) Use EXPLAIN ANALYZE to expect how your execution plan and time will change, and then analyze them.

EXECISE 1 Solution

I. 제공된 코드를 사용해서 랜덤 값 100,000개를 각 테이블에 넣습니다.

1)
```
INSERT INTO log_unpart (date, type, amount)
SELECT ('2024-01-01'::DATE + (random() * 364)::INT) AS date,
CASE WHEN random() < 0.5 THEN 'send' ELSE 'receive' END
AS type,
 round(random() * 1000) AS amount
 FROM generate_series(1, 100000);
```

```
d2019320097=# INSERT INTO log_unpart (date, type, amount) SELECT ('2024-01-01'::DATE + (random() * 364)::INT) AS date
, CASE WHEN random() < 0.5 THEN 'send' ELSE 'receive' END AS type, round(random() * 1000) AS amount FROM generate_ser
ies(1, 100000);
INSERT 0 100000
```

2)
```
INSERT INTO log_part (date, type, amount)
SELECT ('2024-01-01'::DATE + (random() * 364)::INT) AS date,
CASE WHEN random() < 0.5 THEN 'send' ELSE 'receive' END
AS type,
 round(random() * 1000) AS amount
 FROM generate_series(1, 100000);
```

```
d2019320097=# INSERT INTO log_part (date, type, amount) SELECT ('2024-01-01'::DATE + (random() * 364)::INT) AS date,
CASE WHEN random() < 0.5 THEN 'send' ELSE 'receive' END AS type, round(random() * 1000) AS amount FROM generate_serie
s(1, 100000);
INSERT 0 100000
```

랜덤한 날짜, 타입, amount를 가지는 레코드 100,000개가 각 테이블에 추가됩니다.

II. 특정 파티션의 레코드를 출력하는 쿼리를 작성하거나, 각 파티션의 레코드 수를 세는 쿼리를 작성합니다.

1)
```
Select * from log_unpart_q2;
SELECT COUNT(*) AS record_count FROM log_part_q2;
```

이 쿼리를 통해서 별도의 작업 없이도 해당하는 범위의 레코드가 파티션으로 분할되어 들어갔음을 확인할 수 있습니다.

```
d2019320097=# select count(*) from log_part_q1;
 count
-------
 24819
(1개 행)


d2019320097=# select count(*) from log_part_q2;
 count
-------
 25158
(1개 행)
```

EXECISE 1 Solution

III. 특정 분기에서만 레코드를 읽는 쿼리를 작성합니다.

1) SELECT * FROM log_part WHERE date BETWEEN '2024-04-01' AND '2024-06-30';

SELECT * FROM log_unpart WHERE date BETWEEN '2024-04-01' AND '2024-06-30';

이 두 쿼리는 q2 파티션 범위 내에 있는 레코드만 읽습니다.

2) 각 쿼리를 EXPLAIN ANLYZE를 통해 분석하면 log_unpart는 전체 테이블을, log_part는 q2 파티션만 스캔하게 됩니다.
두 경우 모두 seq scan을 사용하나 스캔 수 차이에 의해서 log_part의 쿼리가 더 빠른 실행시간을 보장합니다.

```
d2019320097=# explain analyze
d2019320097=# SELECT * FROM log_part WHERE date BETWEEN '2024-04-01' AND '2024-06-30';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on log_part_q2 log_part  (cost=0.00..11260.92 rows=526928 width=18) (actual time=0.036..102.106 rows=526928 loops=1)
   Filter: ((date >= '2024-04-01'::date) AND (date <= '2024-06-30'::date))
 Planning Time: 0.369 ms
 Execution Time: 132.642 ms
(4개 행)


d2019320097=# explain analyze
d2019320097=# SELECT * FROM log_unpart WHERE date BETWEEN '2024-04-01' AND '2024-06-30';
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on log_unpart  (cost=0.00..23507.00 rows=267011 width=18) (actual time=0.070..167.884 rows=274602 loops=1)
   Filter: ((date >= '2024-04-01'::date) AND (date <= '2024-06-30'::date))
   Rows Removed by Filter: 825398
 Planning Time: 0.148 ms
 Execution Time: 181.255 ms
(5개 행)
```

IV. 여러 분기를 포함하는 범위의 레코드를 읽는 쿼리를 작성합니다.

1) SELECT * FROM Log_part WHERE date BETWEEN '2024-04-01' AND '2024-12-30';

SELECT * FROM Log_unpart WHERE date BETWEEN '2024-04-01' AND '2024-12-30';

이 쿼리는 q2 ~ q4 파티션에 포함된 레코드들을 읽습니다.

2) 각 쿼리를 EXPLAIN ANALYZE를 통해 분석하면
log_unpart는 테이블 전체를 seq scan하여 범위에 해당하는 레코드를 찾습니다.
반면 log_part는 범위에 해당하는 파티션을 먼저 찾고, 그 파티션만을 seq scan하는 table pruning(테이블 프루닝)이 발생합니다.
하지만 테이블 프루닝에 의한 오버헤드가 커 오히려 log_unpart 쪽이 더 빠른 현상이 발생합니다.

```
d2019320097=# explain analyze
d2019320097=# SELECT * FROM log_part WHERE date BETWEEN '2024-04-01' AND '2024-12-30';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Append  (cost=0.00..1984.62 rows=75181 width=18) (actual time=0.027..14.661 rows=75181 loops=1)
   ->  Seq Scan on log_part_q2 log_part_1  (cost=0.00..538.37 rows=25158 width=18) (actual time=0.025..3.172 rows=25158 loops=1)
         Filter: ((date >= '2024-04-01'::date) AND (date <= '2024-12-30'::date))
   ->  Seq Scan on log_part_q3 log_part_2  (cost=0.00..538.84 rows=25189 width=18) (actual time=0.010..2.686 rows=25189 loops=1)
         Filter: ((date >= '2024-04-01'::date) AND (date <= '2024-12-30'::date))
   ->  Seq Scan on log_part_q4 log_part_3  (cost=0.00..531.51 rows=24834 width=18) (actual time=0.008..2.776 rows=24834 loops=1)
         Filter: ((date >= '2024-04-01'::date) AND (date <= '2024-12-30'::date))
 Planning Time: 0.384 ms
 Execution Time: 17.684 ms
(9개 행)
```

```
d2019320097=# explain analyze
d2019320097=# SELECT * FROM Log_unpart WHERE date BETWEEN '2024-04-01' AND '2024-12-30';
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on log_unpart  (cost=0.00..2137.00 rows=74800 width=18) (actual time=0.014..11.329 rows=75151 loops=1)
   Filter: ((date >= '2024-04-01'::date) AND (date <= '2024-12-30'::date))
   Rows Removed by Filter: 24849
 Planning Time: 0.192 ms
 Execution Time: 14.386 ms
(5개 행)
```

Exercise 2

I. Write a query that appends at least 100,000 more records to each table.
(Note that you should not add directly to the partition, but only to the log_part and log_unpart).

1) Execute this query on log_unpart and log_part and compare the execution plan and time using EXPLAIN ANALYZE.

II. Write a query that updates the amount of all records in a specific quarter to amount + 100.

1) Run each query on log_part and log_unpart and use EXPLAIN ANALYZE to compare the execution plan and time.

III. Write a query that deletes all records in a specific quarter.

1) Run query on log_unpart with EXPLAIN ANALYZE.
2) On log_part, There are two ways of queries you can create: (DELETE or DROP TABLE)
If possible, try using EXPLAIN ANALYZE and compare execution plan and time.

(Hint. You can use ALTER TABLE log_part DETACH PARTITION [partition_name];
and DROP TABLE to drop a specific partition).

Exercise 2 Solution

I. Exercise 1에서 사용했던 것과 동일한 쿼리를 사용해서 레코드를 추가할 수 있습니다.

1)
```
INSERT INTO log_unpart (date, type, amount)
SELECT ('2024-01-01'::DATE + (random() * 364)::INT) AS date,
CASE WHEN random() < 0.5 THEN 'send' ELSE 'receive' END AS type,
round(random() * 1000) AS amount
FROM generate_series(1, 100000);
```

2)
```
INSERT INTO log_part (date, type, amount)
SELECT ('2024-01-01'::DATE + (random() * 364)::INT) AS date,
CASE WHEN random() < 0.5 THEN 'send' ELSE 'receive' END AS type,
round(random() * 1000) AS amount
FROM generate_series(1, 100000);
```

3) 두 쿼리를 EXPLAIN ANALYZE를 사용해 비교해보면, log_unpart에 삽입하는 것에 비해 log_part에 삽입할 때 비용이 엇비슷하거나 더 큰 것을 확인할 수 있습니다.

그 이유로는 log_part에 삽입하기 위해서 각 레코드가 어떤 파티션에 해당하는지 확인하는 작업을 한 번 더 거치는 오버헤드가 지속적으로 발생하고, 각 파티션에 인덱스를 계속해서 갱신해야 하기 때문입니다.
즉, 파티션에 자주 삽입이 일어나는 경우에는 파티셔닝이 적합하지 않을 수 있습니다.

II. 특정 파티션 범위를 수정할 때 Log_unpart의 경우에는 조건에 범위를 지정하여 쿼리를 작성하고, log_part는 특정 파티션을 지정하여 업데이트 할 수 있습니다.

1)
```
UPDATE log_unpart
SET amount = amount + 100
WHERE date BETWEEN '2024-01-01' AND '2024-04-01';
```

2)
```
UPDATE log_part_q1
SET amount = amount + 100;
```

3) 두 쿼리를 EXPLAIN ANALYZE를 통해 비교해보면 log_part에서 특정 파티션에 업데이트하는 것이 log_unpart에서 범위 조건을 지정하는 것 보다 빠름을 알 수 있습니다.

Log_unpart의 경우 전체 테이블을 스캔, 조건에 맞는 행을 골라 업데이트 하는 과정을 거치지만, log_part는 상대적으로 작은 크기의 특정 파티션을 모두 업데이트하기만 하면 됩니다.

```
d2019320097=# EXPLAIN ANALYZE
d2019320097-# INSERT INTO log_unpart (date, type, amount) SELECT ('2024-01-01'::DATE + (random() * 364)::INT) AS date, C
ASE WHEN random() < 0.5 THEN 'send' ELSE 'receive' END AS type, round(random() * 1000) AS amount FROM generate_series(1,
 100000);
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
-------------------
 Insert on log_unpart  (cost=0.00..4250.00 rows=0 width=0) (actual time=833.784..833.785 rows=0 loops=1)
   ->  Subquery Scan on "*SELECT*"  (cost=0.00..4250.00 rows=100000 width=78) (actual time=11.929..330.187 rows=100000 l
oops=1)
         ->  Function Scan on generate_series  (cost=0.00..3250.00 rows=100000 width=44) (actual time=11.900..97.330 row
s=100000 loops=1)
 Planning Time: 0.171 ms
 Execution Time: 836.519 ms
(5개 행)
d2019320097=# EXPLAIN ANALYZE
d2019320097-# INSERT INTO log_part (date, type, amount) SELECT ('2024-01-01'::DATE + (random() * 364)::INT) AS date, CAS
E WHEN random() < 0.5 THEN 'send' ELSE 'receive' END AS type, round(random() * 1000) AS amount FROM generate_series(1, 1
00000);
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
-------------------
 Insert on log_part  (cost=0.00..4250.00 rows=0 width=0) (actual time=919.271..919.273 rows=0 loops=1)
   ->  Subquery Scan on "*SELECT*"  (cost=0.00..4250.00 rows=100000 width=78) (actual time=11.146..286.281 rows=100000 l
oops=1)
         ->  Function Scan on generate_series  (cost=0.00..3250.00 rows=100000 width=44) (actual time=11.112..83.294 row
s=100000 loops=1)
 Planning Time: 0.088 ms
 Execution Time: 936.766 ms
(5개 행)
```

```
d2019320097=# EXPLAIN ANALYZE
d2019320097-# UPDATE log_unpart
d2019320097-# SET amount = amount + 100
d2019320097-# WHERE date BETWEEN '2024-01-01' AND '2024-04-01';
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
-----
 Update on log_unpart  (cost=0.00..21239.66 rows=0 width=0) (actual time=2343.796..2343.797 rows=0 loops=1)
   ->  Seq Scan on log_unpart  (cost=0.00..21239.66 rows=224665 width=38) (actual time=62.400..208.254 rows=226833 loops
=1)
         Filter: ((date >= '2024-01-01'::date) AND (date <= '2024-04-01'::date))
         Rows Removed by Filter: 673167
 Planning Time: 2.138 ms
 Execution Time: 2343.842 ms
(6개 행)
d2019320097=# EXPLAIN ANALYZE
d2019320097-# UPDATE log_part_q1
d2019320097-# SET amount = amount + 100;
                                                       QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
--
 Update on log_part_q1  (cost=0.00..5653.96 rows=0 width=0) (actual time=2088.016..2088.017 rows=0 loops=1)
   ->  Seq Scan on log_part_q1  (cost=0.00..5653.96 rows=223997 width=38) (actual time=1.062..83.214 rows=223997 loops=1
)
 Planning Time: 1.508 ms
 Execution Time: 2088.073 ms
(4개 행)
```

Exercise 2 Solution

III. 특정 파티션의 데이터를 모두 삭제하는 경우, log_unpart는 범위 조건을 지정하여 삭제해야합니다. log_part는 DELETE를 통해 특정 파티션의 행들을 삭제하거나, DROP TABLE을 통해 파티션 자체를 삭제할 수 있습니다.

1)
```
DELETE FROM log_unpart
WHERE date BETWEEN '2024-04-01' AND '2024-06-30';

DELETE FROM log_part_q2;
```

2)
```
ALTER TABLE log_part DETACH PARTITION log_part_q2;
DROP TABLE log_part_q2;
```

3) DELETE를 통해 삭제하는 경우, log_unpart는 모든 행을 스캔 후 조건에 맞는 행을 삭제하므로 log_part 에서 특정 파티션 전체 행을 삭제하는 것에 비해 더 오랜 시간이 걸립니다.

DROP TABLE 을 통해 삭제하는 경우, 해당 파티션을 더 이상 이용할 수 없지만 대량의 데이터도 일순간에 삭제할 수 있으며 저장공간의 여유를 만들 수 있습니다.

```
d2019320097=# EXPLAIN ANALYZE
d2019320097-# DELETE FROM log_unpart
d2019320097-# WHERE date BETWEEN '2024-04-01' AND '2024-06-30';
                                                                  QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------------
--
 Delete on log_unpart  (cost=0.00..20678.00 rows=0 width=0) (actual time=765.390..765.391 rows=0 loops=1)
   ->  Seq Scan on log_unpart  (cost=0.00..20678.00 rows=226776 width=6) (actual time=0.021..119.819 rows=224928 loops=1
)
         Filter: ((date >= '2024-04-01'::date) AND (date <= '2024-06-30'::date))
         Rows Removed by Filter: 675072
 Planning Time: 0.291 ms
 Execution Time: 765.432 ms
(6개 행)
```

```
d2019320097=# EXPLAIN ANALYZE
d2019320097-# DELETE FROM log_part_q2;
                                                          QUERY PLAN


-----------------------------------------------------------------------------------------------------------------------------------------------
-
 Delete on log_part_q2  (cost=0.00..4088.63 rows=0 width=0) (actual time=240.268..240.270 rows=0 loops=1)
   ->  Seq Scan on log_part_q2  (cost=0.00..4088.63 rows=249763 width=6) (actual time=0.012..30.484 rows=249763 loops=1)
 Planning Time: 1.166 ms
 Execution Time: 240.398 ms
(4개 행)
```

```
d2019320097=# ALTER TABLE log_part DETACH PARTITION log_part_q2;
ALTER TABLE
d2019320097=# DROP TABLE log_part_q2;
DROP TABLE
d2019320097=#
```

Table Lock modes

- In postgresql, tables have different levels of locks depending on what happens in a transaction.

  Here are the locks you need to know for this lab.

Lock mode	Description
SHARE LOCK	can SELECT from another transaction but not change the data.
ROW EXCLUSIVE LOCK	Used for data changes (INSERT, UPDATE, DELETE). Other transaction cannot modify this row.
ACCESS EXCLUSIVE LOCK	Block all other operations(even SELECT). Use to modify the entire table or metadata. Can obtain certain operation (LOCK TABLE, TRUNCATE, ALTER TABLE, CREATE INDEX, ANALYZE…)

- Following code shows the information of locks.

```
SELECT pid, relation::regclass AS table_name, mode, granted
FROM pg_locks
WHERE relation IS NOT NULL;
```

Exercise 3

I. Run two psql shells in separate windows. Start a transaction, write a query that modifies data in one log_part partition, and wait without committing.

1) Look at what locks the parent table log_part and the selected partition have on each and explain why.

2) Write a query to get a different kind of lock and explain why.

II. Suppose you want to make some Scenarios to perform the same operations on the log_part and log_unpart tables.
Run two transactions for each table in parallel.
ex) t1, t2 for log_part and t3, t4 for log_unpart

1) Create scenario and transaction that causes a lock conflict on log_part but runs without a problem on log_unpart.
(Note that you don't need to use the same query in transactions if the result of the operation is the same.)

2) Create scenario and transaction transactions that runs fine on log_part but causes a lock conflict on log_unpart.

Exercise 3 Solution

I. log_part의 특정 데이터를 수정하는 트랜잭션을 만듭니다.

1)
```
BEGIN;
UPDATE log_part
SET amount = amount * 2
WHERE date = '2024-01-10';
```

이 쿼리는 q1 파티션에 있는 데이터를 업데이트합니다.

2) 이때 다른 윈도우에서 테이블에 걸린 락을 확인합니다.
Log_part와 log_part_q1 둘 다에 ROW EXCLUSIVE LOCK이 걸려있음을 확인할 수 있습니다.
행 단위의 수정이 발생하면서 다른 트랜잭션이 동일한 행을 수정할 수 없도록 ROW EXCLUSIVE LOCK을 획득합니다.

II. 다른 종류의 락을 획득하기 위해서는 다른 종류의 트랜잭션을 만들 수 있습니다.

1)
```
BEGIN;
TRUNCATE log_part_q1;
```

이 쿼리는 q1 파티션의 모든 데이터를 한번에 삭제합니다.

2) 마찬가지로 다른 윈도우에서 테이블에 걸린 락을 확인하면 SHARE LOCK과 ACCESS EXCLUSIVE LOCK 이 동시에 걸려있음을 볼 수 있습니다.
테이블의 모든 데이터를 삭제하기 위한 ACCESS EXCLUSIVE LOCK과 인덱스에 해당하는 데이터를 접근하기 위한 SHARE LOCK이 동시에 필요하기 때문입니다.

```
d2019320097=# begin;
BEGIN
d2019320097=*# update log_part
d2019320097-*# set amount = amount * 2
d2019320097-*# where date = '2024-01-10';
UPDATE 2864
d2019320097=*#

d2019320097=# SELECT pid, relation::regclass AS table_name, mode, grante
d FROM pg_locks WHERE relation IS NOT NULL;
  pid   |    table_name     |       mode       | granted
--------+-------------------+------------------+---------
 38124 | pg_locks          | AccessShareLock  | t
 54976 | log_part_q1_pkey  | RowExclusiveLock | t
 54976 | log_part_q1       | RowExclusiveLock | t
 54976 | log_part_pkey     | RowExclusiveLock | t
 54976 | log_part          | RowExclusiveLock | t
(5개 행)
```

```
d2019320097=# begin;
BEGIN
d2019320097=*# TRUNCATE log_part_q1;
TRUNCATE TABLE
d2019320097=*#

d2019320097=# SELECT pid, relation::regclass AS table_name, mode, grante
d FROM pg_locks WHERE relation IS NOT NULL;
  pid   |           table_name           |        mode         | granted
--------+--------------------------------+---------------------+---------
 38124 | pg_locks                       | AccessShareLock     | t
 54976 | pg_toast.pg_toast_34515_index  | AccessExclusiveLock | t
 54976 | log_part_q1_pkey               | AccessExclusiveLock | t
 54976 | log_part_q1                    | ShareLock           | t
 54976 | log_part_q1                    | AccessExclusiveLock | t
 54976 | pg_toast.pg_toast_34515        | ShareLock           | t
 54976 | pg_toast.pg_toast_34515        | AccessExclusiveLock | t
(7개 행)
```

Exercise 3 Solution

III. Log_part에서는 실행되지 못하지만, log_unpart에서는 정상적으로 실행 가능한 시나리오를 가정합니다.

1분기의 로그를 모두 삭제하고 새로운 1분기의 로그를 추가하는 시나리오를 가정해보겠습니다.

IV. Log_part에서

T1
BEGIN;
TRUNCATE log_part_q1;

T1 실행 후,

T2
BEGIN;
INSERT INTO log_part (date, type, amount) values ('2024-01-01', 'send', 100);

이 경우 log_part_q1 테이블에 대한 ACCESS EXCLUSIVE LOCK을 T1 이 가지고 있으므로, T2에서 실행되는 log_part_q1에 대한 삽입은 실행될 수 없습니다.

V. Log_unpart의 경우

T1
BEGIN;
DELETE FROM log_unpart
WHERE date BETWEEN '2024-01-01' AND '2024-03-31';

T1 실행 후

T2
BEGIN;
INSERT INTO log_unpart(date, type, amount) VALUES('2024-01-01', 'send', 100);

이 경우 T1에서 ROW EXCLUSIVE LOCK을 획득, 조건에 맞는 행들을 삭제한 후 락을 해제하므로 T2에서 삽입이 아무런 문제 없이 정상적으로 진행됩니다.

```
d2019320097=# BEGIN;
BEGIN
d2019320097=*# TRUNCATE log_part_q1;
TRUNCATE TABLE
d2019320097=*#
```

```
d2019320097=# BEGIN;
BEGIN
d2019320097=*# INSERT INTO log_part(date, type, amount) VALUES ('2024-01
-01', 'send', 100);
```

```
d2019320097=# BEGIN;
BEGIN
d2019320097=*# DELETE FROM log_unpart
d2019320097-*# WHERE date BETWEEN '2024-01-01' AND '2024-03-31';
DELETE 249059
d2019320097=*#
```

```
d2019320097=# BEGIN;
BEGIN
d2019320097=*# INSERT INTO log_unpart(date, type, amount) VALUES ('2024-
01-01', 'send', 100);
INSERT 0 1
d2019320097=*#
```

Exercise 3 Solution

VI. 반대로, log_part에서는 문제가 없지만 log_unpart에서는 문제가 발생하는 시나리오를 만듭니다.

VII. 예를 들어, 1분기 로그에 대하여 date 에 대한 index를 추가하고 싶다고 가정합니다.
이때 다른 트랜잭션에서는 당일의 로그를 기록합니다.

Log_part의 경우, log_part_q1 테이블 하나에 인덱스를 추가하면 됩니다.

T1
```
BEGIN;
CREATE INDEX ON log_part_q1(date);
```

T2
```
BEGIN;
INSERT INTO log_part(date, type, amount) VALUES ('2024-12-09', 'send', 100);
```

이때 T1에서 log_part_q1에 대한 ACCESS EXCLUSIVE LOCK을 획득하지만 T2에서 삽입이 log_part_q4에 일어나므로 아무런 문제가 발생하지 않습니다.
추가적으로 앞으로의 로그 삽입, 삭제에 인덱스에 의한 추가적인 오버헤드도 발생하지 않을 것입니다.

VIII. Log_unpart의 경우, 특정 파티션에만 인덱스를 걸 수 없으므로 전체 테이블에 인덱스를 걸어야 합니다.

T1
```
BEGIN;
CREATE INDEX ON log_unpart(date);
```

T2
```
BEGIN;
INSERT INTO log_unpart(date, type, amount) VALUES('2024-12-09', 'send', 100);
```

이때 T1에서 log_unpart 테이블 전체에 인덱스를 걸기 위해 ACCESS EXCLUSIVE LOCK을 획득합니다. 따라서 T2의 삽입은 진행되지 못합니다.
추가적으로 추후의 로그의 삽입, 삭제에 있어서도 인덱스에 의한 추가적인 오버헤드가 발생할 것입니다.

```
d2019320097=# BEGIN;
BEGIN
d2019320097=*# CREATE INDEX on log_part_q1(date);
CREATE INDEX
d2019320097=*#
d2019320097=# BEGIN;
BEGIN
d2019320097=*# INSERT INTO log_part(date, type, amount) VALUES ('2024-12
-09', 'send', 100);
INSERT 0 1
d2019320097=*#
```

```
d2019320097=# BEGIN;
BEGIN
d2019320097=*# CREATE INDEX on log_unpart(date);
CREATE INDEX
d2019320097=*#
```

```
d2019320097=# BEGIN;
BEGIN
d2019320097=*# INSERT INTO log_unpart(date, type, amount) VALUES ('2024-
12-09', 'send', 100);
```