Напредни бази на податоци

Фаза 5 – Напредна тема – Партиционирање

Проект: Eventix

Членови на тимот:

- Сандра Ацевска 231070 – координатор
- Филип Ѓоргиев 231122
- Ивана Костовска 232007

Зошто е избрана темата?

За напредната тема е избрано партиционирање на бази на податоци. Главната причина за овој избор е големината на табелата TICKET која содржи 10,000,000 редови и е меѓу најголемите табели во системот. Со растот на апликацијата Eventix, оваа табела континуирано се зголемува со секој нов продаден тикет, па навременото партиционирање е клучно за одржување на перформансите на системот.

Потреба од партиционирање и употреба во системот

Во Eventix корисниците секојдневно извршуваат операции кои директно ја користат табелата TICKET:

- Купување тикет – при секое купување се вметнува нов запис во TICKET
- Преглед на историја на нарачки – корисникот ги прегледува своите тикети филтрирани по датум на настан
- Генерирање финансиски извештаи – се пресметуваат приходи по настан и период

Без партиционирање, секој од овие прашалници мора да ја скенира целата табела со 10,000,000 редови без разлика на тоа колку резултати се очекуваат. Тоа го прави системот бавен и нескалабилен.

Кој аспект од апликацијата се подобрува?

Партиционирањето на табелата TICKET по колоната event_start_date го подобрува аспектот на перформанси при пребарување на тикети по датум на настан. Конкретно, наместо скенирање на целата табела со 10,000,000 редови, PostgreSQL скенира само партицијата која одговара на бараниот временски период.

Ова е особено важно за системот бидејќи најчестите операции се поврзани со конкретни временски периоди. Со партиционирањето, овие операции стануваат до 10x побрзи само со Partition Pruning, а до 5000x побрзи во комбинација со индекси.

Причини за избор на RANGE партиционирање

- Датумот е континуирана вредност со природни интервали
- Секоја година добива своја партиција со ~1.7M редови
- Partition Pruning ефикасно работи кога се филтрира по датум

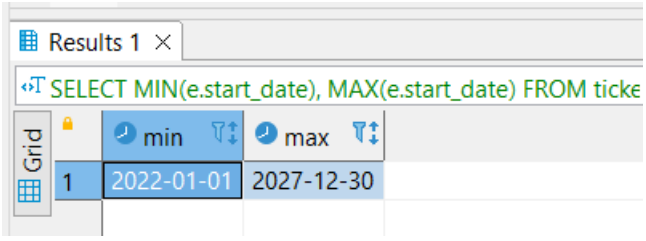
Креирање на партиционираната табела

Главната табела се креира со PARTITION BY RANGE (event_start_date) клаузула која укажува дека партиционирањето ќе биде по колоната event_start_date со RANGE тип. Колоната event_start_date е додадена специјално за партиционирање бидејќи оригиналната табела TICKET нема директна датумска колона.

```
CREATE TABLE ticket_partitioned (
    id              BIGSERIAL   NOT NULL,
    code            VARCHAR(50) NOT NULL,
    status          VARCHAR(20) NOT NULL,
    TICKET_TYPEid   BIGINT NOT NULL,
    USER_ORDERid    BIGINT NOT NULL,
    SEATid          BIGINT,
    APP_USERid      BIGINT NOT NULL,
    EVENTid         BIGINT NOT NULL,
    HALLid          BIGINT NOT NULL,
    event_start_date DATE NOT NULL
) PARTITION BY RANGE (event_start_date);
```

Партиционираната табела е креирана паралелно со постоечката табела ticket за да може да се споредат перформансите без да се наруши постоечкиот систем.

Партиции



Опсегот на датуми во табелата е од 2022-01-01 до 2027-12-30, па се креирани 6 партиции – по една за секоја година.

За секоја година се креира посебна партиција со RANGE вредности од почетокот до крајот на годината. Дополнително се креира default партиција која ги фаќа сите вредности кои не се покриени со останатите партиции.

```
CREATE TABLE ticket_2022 PARTITION OF ticket_partitioned
    FOR VALUES FROM ('2022-01-01') TO ('2023-01-01');

CREATE TABLE ticket_2023 PARTITION OF ticket_partitioned
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE ticket_2024 PARTITION OF ticket_partitioned
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

CREATE TABLE ticket_2025 PARTITION OF ticket_partitioned
    FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');

CREATE TABLE ticket_2026 PARTITION OF ticket_partitioned
    FOR VALUES FROM ('2026-01-01') TO ('2027-01-01');

CREATE TABLE ticket_2027 PARTITION OF ticket_partitioned
    FOR VALUES FROM ('2027-01-01') TO ('2028-01-01');

CREATE TABLE ticket_default PARTITION OF ticket_partitioned
    DEFAULT;
```

Полнење на податоци

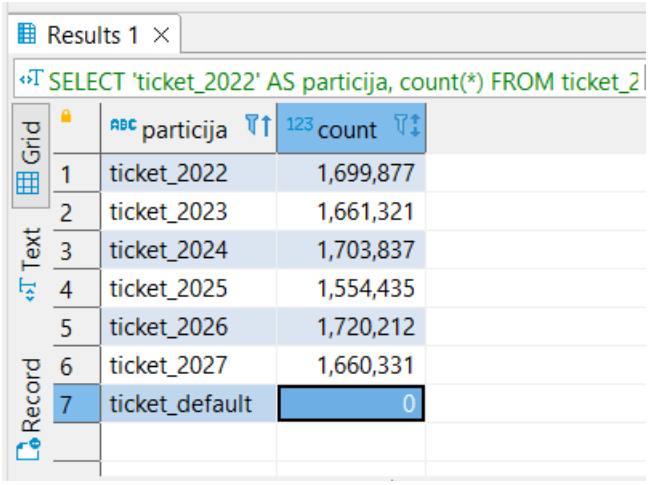
Податоците од постоечката табела ticket се префрлаат во партиционираната табела со JOIN на табелата EVENT за да се добие датумот на настанот. Колоната event_start_date се пополнува од колоната start_date на табелата EVENT.

```
INSERT INTO ticket_partitioned
    (id, code, status, ticket_typeid, user_orderid,
     seatid, app_userid, eventid, hallid, event_start_date)
SELECT
    t.id, t.code, t.status, t.ticket_typeid, t.user_orderid,
    t.seatid, t.app_userid, t.eventid, t.hallid, e.start_date
FROM ticket t
JOIN event e ON e.id = t.eventid;
```

Резултат: 10,000,013 редови префрлени успешно.

Распределба на податоците по партиции

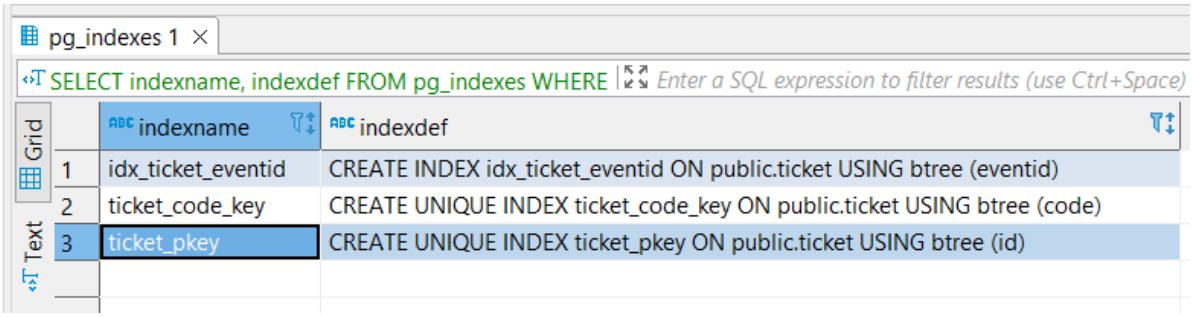
По полнењето, податоците се распределени на следниот начин:



	particija	count
1	ticket_2022	1,699,877
2	ticket_2023	1,661,321
3	ticket_2024	1,703,837
4	ticket_2025	1,554,435
5	ticket_2026	1,720,212
6	ticket_2027	1,660,331
7	ticket_default	0

Податоците се рамномерно распределени по партиции со приближно 1.7M редови по година. ticket_default е празна што потврдува дека сите вредности се покриени со дефинираните партиции.

Индексирање на партициите



	indexname	indexdef
1	idx_ticket_eventid	CREATE INDEX idx_ticket_eventid ON public.ticket USING btree (eventid)
2	ticket_code_key	CREATE UNIQUE INDEX ticket_code_key ON public.ticket USING btree (code)
3	ticket_pkey	CREATE UNIQUE INDEX ticket_pkey ON public.ticket USING btree (id)

Индексите не се наследуваат автоматски при партиционирање и треба да се креираат посебно за секоја партиција. По проверка за кои индекси ги има оригиналната табела TICKET, видливи се:

- ticket_pkey – PRIMARY KEY индекс, автоматски креиран за гаранција дека секој тикет има уникатен id
- ticket_code_key – UNIQUE индекс на code, автоматски креиран поради unique constraint на колоната code за гаранција дека секој тикет има уникатен код
- idx_ticket_eventid – индекс на eventid, рачно креиран при оптимизацијата на погледите за забрзано пребарување по eventid.

За партицинираната табела:

- ticket_pkey и ticket_code_key – не се потребни бидејќи партицинираната табела е само за демонстрација и нема FK constraints
- idx_ticket_eventid – треба да се креира на секоја партиција бидејќи е важен за перформанси

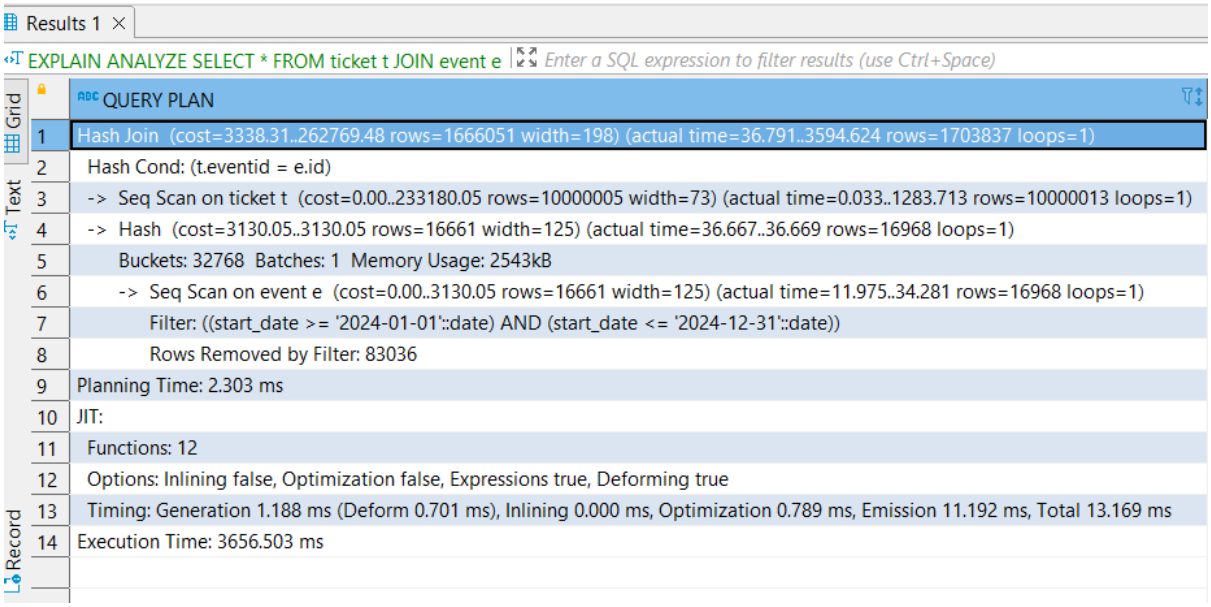
```sql
CREATE INDEX idx_ticket_2022_eventid ON ticket_2022(eventid);
CREATE INDEX idx_ticket_2023_eventid ON ticket_2023(eventid);
CREATE INDEX idx_ticket_2024_eventid ON ticket_2024(eventid);
CREATE INDEX idx_ticket_2025_eventid ON ticket_2025(eventid);
CREATE INDEX idx_ticket_2026_eventid ON ticket_2026(eventid);
CREATE INDEX idx_ticket_2027_eventid ON ticket_2027(eventid);
```

Индексите на партициите се помали и побрзи бидејќи покриваат само ~1.7M редови наместо 10M.

Тестирање на перформансите

Тест 1 – Без партиционирање

```sql
EXPLAIN ANALYZE SELECT * FROM ticket t
JOIN event e ON e.id = t.eventid
WHERE e.start_date BETWEEN '2024-01-01' AND '2024-12-31';
```

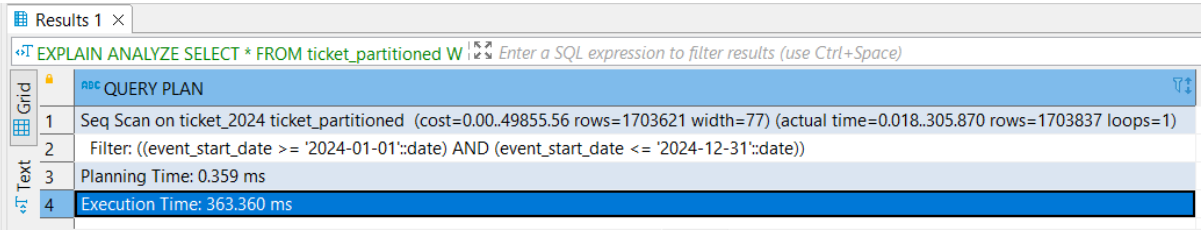


PostgreSQL врши Hash Join помеѓу табелите ticket и event. За да ги најде тикетите за 2024 година, прво мора да врши Seq Scan на целата табела ticket со 10,000,000 редови, а потоа да ги спои со резултатите од Seq Scan на табелата event. Ова е бавно бидејќи нема начин да се ограничи пребарувањето без партиционирање.

Execution Time: 3656ms

Тест 2 – Со партиционирање, без индекси (само Partition Pruning)

```sql
EXPLAIN ANALYZE SELECT * FROM ticket_partitioned
WHERE event_start_date BETWEEN '2024-01-01' AND '2024-12-31';
```



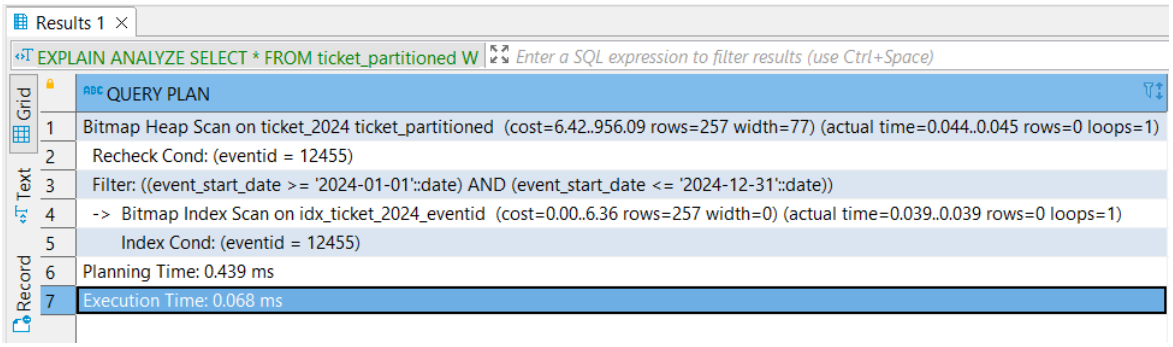
PostgreSQL применува Partition Pruning и врши Seq Scan само на партицијата ticket_2024 со 1,703,837 редови. Останатите 5 партиции се целосно прескокнати бидејќи не можат да содржат редови со датум во 2024 година.

Само со Partition Pruning, без никакви дополнителни индекси, времето се намалува од 3656ms на 363ms – приближно 10x побрзо.

Execution Time: 363ms

Тест 3 – Со партиционирање и индекси (Partition Pruning + Index Scan)

```
EXPLAIN ANALYZE SELECT * FROM ticket_partitioned
WHERE event_start_date BETWEEN '2024-01-01' AND '2024-12-31'
AND eventid = 12455;
```



PostgreSQL комбинира два механизми:

• Partition Pruning – прво оди директно во партицијата ticket_2024

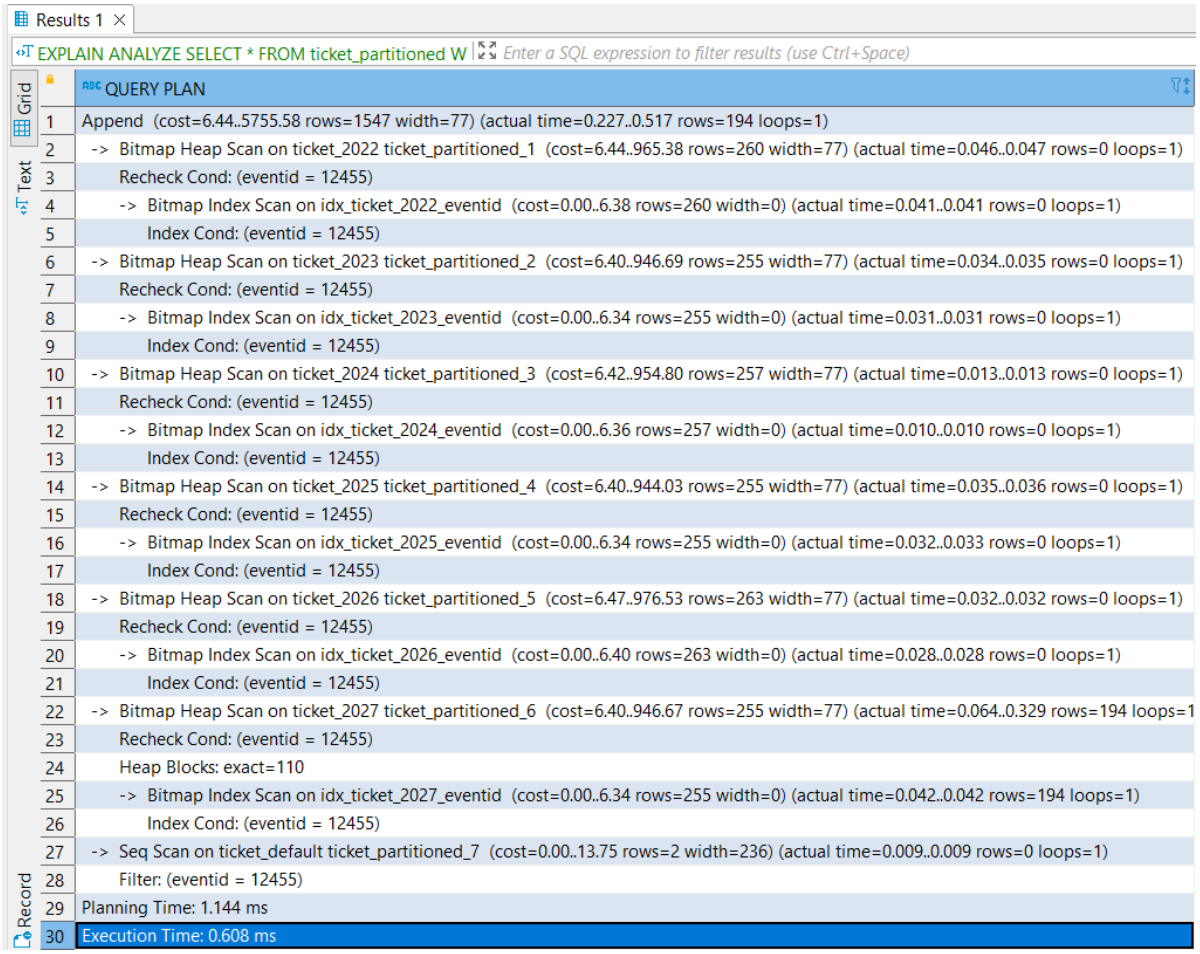
• Bitmap Index Scan on idx_ticket_2024_eventid – потоа го користи индексот за да ги најде само тикетите за настанот со id 12455 без да скенира 1.7M редови.

На тој начин наместо скенирање на 10M редови, PostgreSQL директно ги наоѓа само релевантните редови. Ова е најефикасниот начин на пребарување при партиционирање.

Execution Time: 0.068ms

Тест 4 – No Partition Pruning (без филтер на датум)

```
EXPLAIN ANALYZE SELECT * FROM ticket_partitioned
WHERE eventid = 12455;
```



Results 1

EXPLAIN ANALYZE SELECT * FROM ticket_partitioned W | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text | Record

	QUERY PLAN
1	Append (cost=6.44..5755.58 rows=1547 width=77) (actual time=0.227..0.517 rows=194 loops=1)
2	-> Bitmap Heap Scan on ticket_2022 ticket_partitioned_1 (cost=6.44..965.38 rows=260 width=77) (actual time=0.046..0.047 rows=0 loops=1)
3	Recheck Cond: (eventid = 12455)
4	-> Bitmap Index Scan on idx_ticket_2022_eventid (cost=0.00..6.38 rows=260 width=0) (actual time=0.041..0.041 rows=0 loops=1)
5	Index Cond: (eventid = 12455)
6	-> Bitmap Heap Scan on ticket_2023 ticket_partitioned_2 (cost=6.40..946.69 rows=255 width=77) (actual time=0.034..0.035 rows=0 loops=1)
7	Recheck Cond: (eventid = 12455)
8	-> Bitmap Index Scan on idx_ticket_2023_eventid (cost=0.00..6.34 rows=255 width=0) (actual time=0.031..0.031 rows=0 loops=1)
9	Index Cond: (eventid = 12455)
10	-> Bitmap Heap Scan on ticket_2024 ticket_partitioned_3 (cost=6.42..954.80 rows=257 width=77) (actual time=0.013..0.013 rows=0 loops=1)
11	Recheck Cond: (eventid = 12455)
12	-> Bitmap Index Scan on idx_ticket_2024_eventid (cost=0.00..6.36 rows=257 width=0) (actual time=0.010..0.010 rows=0 loops=1)
13	Index Cond: (eventid = 12455)
14	-> Bitmap Heap Scan on ticket_2025 ticket_partitioned_4 (cost=6.40..944.03 rows=255 width=77) (actual time=0.035..0.036 rows=0 loops=1)
15	Recheck Cond: (eventid = 12455)
16	-> Bitmap Index Scan on idx_ticket_2025_eventid (cost=0.00..6.34 rows=255 width=0) (actual time=0.032..0.033 rows=0 loops=1)
17	Index Cond: (eventid = 12455)
18	-> Bitmap Heap Scan on ticket_2026 ticket_partitioned_5 (cost=6.47..976.53 rows=263 width=77) (actual time=0.032..0.032 rows=0 loops=1)
19	Recheck Cond: (eventid = 12455)
20	-> Bitmap Index Scan on idx_ticket_2026_eventid (cost=0.00..6.40 rows=263 width=0) (actual time=0.028..0.028 rows=0 loops=1)
21	Index Cond: (eventid = 12455)
22	-> Bitmap Heap Scan on ticket_2027 ticket_partitioned_6 (cost=6.40..946.67 rows=255 width=77) (actual time=0.064..0.329 rows=194 loops=1
23	Recheck Cond: (eventid = 12455)
24	Heap Blocks: exact=110
25	-> Bitmap Index Scan on idx_ticket_2027_eventid (cost=0.00..6.34 rows=255 width=0) (actual time=0.042..0.042 rows=194 loops=1)
26	Index Cond: (eventid = 12455)
27	-> Seq Scan on ticket_default ticket_partitioned_7 (cost=0.00..13.75 rows=2 width=236) (actual time=0.009..0.009 rows=0 loops=1)
28	Filter: (eventid = 12455)
29	Planning Time: 1.144 ms
30	Execution Time: 0.608 ms

Кога прашалникот нема филтер на колоната event_start_date која е partition key, PostgreSQL не може да примени Partition Pruning. Затоа врши Append операција и ги скенира сите 6 партиции паралелно со Bitmap Index Scan на индексот на секоја партиција.

Иако индексите помагаат и времето е само 0.608ms, тоа е сепак побавно отколку Тест 3 (0.068ms) каде се комбинираат Partition Pruning и индекс. Ова е пример за No Partition Pruning – честа грешка при користење на партиционирање.

Execution Time: 0.608ms

РЕЗИМЕ НА РЕЗУЛТАТИ

Тест 1 – Без партиционирање

Без партиционирање, PostgreSQL мора да ја скенира целата табела ticket со 10,000,000 редови и да ги спои со табелата event преку Hash Join за да ги најде тикетите за 2024 година.

Execution Time: 3656ms

Тест 2 – Само Partition Pruning, без индекси

Со партиционирање и филтер по event_start_date, PostgreSQL применува Partition Pruning и ја скенира само партицијата ticket_2024 со 1,703,837 редови. Останатите 5 партиции се целосно прескокнати. Само со Partition Pruning, без никакви дополнителни индекси, времето се намалува од 3,656ms на 363ms - приближно 10х побрзо.

Execution Time: 363ms

Тест 3 – Partition Pruning + индекс

Со додавање на филтер по eventid покрај датумот, PostgreSQL комбинира два механизми. Прво со Partition Pruning оди директно во ticket_2024, а потоа наместо Seq Scan на 1,703,837 редови, го користи индексот idx_ticket_2024_eventid и директно ги наоѓа само тикетите за тој настан.

Комбинацијата на Partition Pruning и индекс го намалува времето од 363ms на 0.068ms.

Execution Time: 0.068ms

Тест 4 – No Partition Pruning

Кога прашалникот филтрира само по eventid без филтер на event_start_date, PostgreSQL не може да примени Partition Pruning и не знае во која партиција да бара. Затоа врши Append операција и ги скенира сите 6 партиции паралелно.

Иако го користи индексот на секоја партиција, времето е 0.608ms — приближно 9х побавно од Тест 3 каде се применува и Partition Pruning. Ова јасно покажува дека индексите сами по себе не се доволни и дека партиционирањето дава најдобри резултати само кога прашалникот филтрира по partition key.

Execution Time: 0.608ms

ЗАКЛУЧОК

Партиционирањето на табелата TICKET по колоната event_start_date со RANGE партиционирање се покажа како ефикасна техника за подобрување на перформансите. Поделбата на 10,000,000 редови во 6 партиции по година овозможи PostgreSQL да ја скенира само релевантната партиција наместо целата табела.

Најголемо подобрување се постигнува кога прашалникот филтрира по partition key (event_start_date) – Partition Pruning го намалува просторот за пребарување, а индексите го забрзуваат пребарувањето внатре во партицијата. Комбинацијата на двата механизми го намалува времето на извршување од 3656ms на 0.068ms.

Важно е да се напомене дека партиционирањето не е универзално решение. Кога прашалникот не филтрира по partition key, PostgreSQL мора да ги скенира сите партиции и придобивките се намалуваат. Затоа при проектирање на партиционирана табела треба внимателно да се избере partition key кој одговара на најчестите прашалници во системот.