

PostgreSQL - fetch the rows which have the Max value for a column in each GROUP BY group

Asked 15 years, 10 months ago Modified 1 year, 5 months ago Viewed 193k times



171



I'm dealing with a Postgres table (called "lives") that contains records with columns for time_stamp, usr_id, transaction_id, and lives_remaining. I need a query that will give me the most recent lives_remaining total for each usr_id

1. There are multiple users (distinct usr_id's)
2. time_stamp is not a unique identifier: sometimes user events (one by row in the table) will occur with the same time_stamp.
3. trans_id is unique only for very small time ranges: over time it repeats
4. remaining_lives (for a given user) can both increase and decrease over time

example:

time_stamp	lives_remaining	usr_id	trans_id
07:00	1	1	1
09:00	4	2	2
10:00	2	3	3
10:00	1	2	4
11:00	4	1	5
11:00	3	1	6
13:00	3	3	1

As I will need to access other columns of the row with the latest data for each given usr_id, I need a query that gives a result like this:

time_stamp	lives_remaining	usr_id	trans_id
11:00	3	1	6
10:00	1	2	4
13:00	3	3	1

As mentioned, each usr_id can gain or lose lives, and sometimes these timestamped events occur so close together that they have the same timestamp! Therefore this query won't work:

```
SELECT b.time_stamp,b.lives_remaining,b.usr_id,b.trans_id FROM
  (SELECT usr_id, max(time_stamp) AS max_timestamp
   FROM lives GROUP BY usr_id ORDER BY usr_id) a
JOIN lives b ON a.max_timestamp = b.time_stamp
```

Instead, I need to use both `time_stamp` (first) and `trans_id` (second) to identify the correct row. I also then need to pass that information from the subquery to the main query that will provide the data for the other columns of the appropriate rows. This is the hacked up query that I've gotten to work:

```
SELECT b.time_stamp,b.lives_remaining,b.usr_id,b.trans_id FROM
  (SELECT usr_id, max(time_stamp || '*' || trans_id)
   AS max_timestamp_transid
   FROM lives GROUP BY usr_id ORDER BY usr_id) a
JOIN lives b ON a.max_timestamp_transid = b.time_stamp || '*' || b.trans_id
ORDER BY b.usr_id
```

Okay, so this works, but I don't like it. It requires a query within a query, a self join, and it seems to me that it could be much simpler by grabbing the row that MAX found to have the largest timestamp and trans_id. The table "lives" has tens of millions of rows to parse, so I'd like this query to be as fast and efficient as possible. I'm new to RDBM and Postgres in particular, so I know that I need to make effective use of the proper indexes. I'm a bit lost on how to optimize.

I found a similar discussion [here](#). Can I perform some type of Postgres equivalent to an Oracle analytic function?

Any advice on accessing related column information used by an aggregate function (like MAX), creating indexes, and creating better queries would be much appreciated!

P.S. You can use the following to create my example case:

```
create TABLE lives (time_stamp timestamp, lives_remaining integer,
                     usr_id integer, trans_id integer);
insert into lives values ('2000-01-01 07:00', 1, 1, 1);
insert into lives values ('2000-01-01 09:00', 4, 2, 2);
insert into lives values ('2000-01-01 10:00', 2, 3, 3);
insert into lives values ('2000-01-01 10:00', 1, 2, 4);
insert into lives values ('2000-01-01 11:00', 4, 1, 5);
insert into lives values ('2000-01-01 11:00', 3, 1, 6);
insert into lives values ('2000-01-01 13:00', 3, 3, 1);
```

sql postgresql query-optimization cbo cost-based-optimizer

Share Follow

edited Apr 19, 2022 at 11:09



Ciro Santilli
OurBigBook.com

380k ● 117 ● 1.3k ● 1.1k

asked Feb 25, 2009 at 16:37




Joshua Berry
2,320 ● 3 ● 21 ● 24

Josh, you may not like the fact that the query self-joins etc., but that's OK as far as the RDBMS is concerned. – [vladr](#) Feb 26, 2009 at 8:10

- 1 What the self-join will actually end up translating to is a simple index mapping, where the inner SELECT (the one with MAX) scans the index throwing away irrelevant entries, and where the outer SELECT just grabs the rest of the columns from the table corresponding to the narrowed-down index. – [vladr](#) Feb 26, 2009 at 8:11

Vlad, thanks for the tips and explanation. It's opened my eyes to how to begin understanding the inner workings of the database and how to optimize queries. Quassnoi, thanks for the great query and tip on primary key; Bill too. Very helpful. – [Joshua Berry](#) Feb 26, 2009 at 16:55

thank you for showing me how to get a `MAX` `BY` 2 columns! – [user1382306](#) Oct 9, 2013 at 22:22 

- 1 Possible duplicate of [Fetch the row which has the Max value for a column](#) – [mlt](#) Jan 21, 2016 at 21:18

10 Answers

Sorted by: Highest score (default) 



I would propose a clean version based on `DISTINCT ON` (see [docs](#)):

210



```
SELECT DISTINCT ON (usr_id)
    time_stamp,
    lives_remaining,
    usr_id,
    trans_id
FROM lives
ORDER BY usr_id, time_stamp DESC, trans_id DESC;
```



Share Follow


edited Jul 13, 2015 at 10:31

answered Jun 11, 2015 at 8:40



[Marco](#)

3,381  5  29  29

This seemed to work for me on my slightly different application where nothing else would. Definitely should be raised up for more visibility. – [Jim Factor](#) Oct 26, 2017 at 6:33 



On a table with 158k pseudo-random rows (`usr_id` uniformly distributed between 0 and 10k, `trans_id` uniformly distributed between 0 and 30),

123



By query cost, below, I am referring to Postgres' cost based optimizer's cost estimate (with Postgres' default `xxx_cost` values), which is a weighed function estimate of required I/O and CPU resources; you can obtain this by firing up PgAdminIII and running "Query/Explain (F7)" on the query with "Query/Explain options" set to "Analyze"





- Quassnoy's query has a cost estimate of 745k (!), and completes in 1.3 seconds (given a compound index on (`usr_id`, `trans_id`, `time_stamp`))
- Bill's query has a cost estimate of 93k, and completes in 2.9 seconds (given a compound index on (`usr_id`, `trans_id`))
- **Query #1 below** has a cost estimate of 16k, and completes in 800ms (given a compound index on (`usr_id`, `trans_id`, `time_stamp`))
- **Query #2 below** has a cost estimate of 14k, and completes in 800ms (given a compound function index on (`usr_id`, `EXTRACT(EPOCH FROM time_stamp)`, `trans_id`))
 - this is Postgres-specific
- **Query #3 below** (Postgres 8.4+) has a cost estimate and completion time comparable to (or better than) query #2 (given a compound index on (`usr_id`, `time_stamp`, `trans_id`)); it has the advantage of scanning the `lives` table only once and, should you temporarily increase (if needed) [work_mem](#) to accommodate the sort in memory, it will be by far the fastest of all queries.

All times above include retrieval of the full 10k rows result-set.

Your goal is minimal cost estimate *and* minimal query execution time, with an emphasis on estimated cost. Query execution can depend significantly on runtime conditions (e.g. whether relevant rows are already fully cached in memory or not), whereas the cost estimate is not. On the other hand, keep in mind that cost estimate is exactly that, an estimate.

The best query execution time is obtained when running on a dedicated database without load (e.g. playing with pgAdminIII on a development PC.) Query time will vary in production based on actual machine load/data access spread. When one query appears slightly faster (<20%) than the other but has a *much* higher cost, it will generally be wiser to choose the one with higher execution time but lower cost.

When you expect that there will be no competition for memory on your production machine at the time the query is run (e.g. the RDBMS cache and filesystem cache won't be thrashed by concurrent queries and/or filesystem activity) then the query time you obtained in standalone (e.g. pgAdminIII on a development PC) mode will be representative. If there is contention on the production system, query time will degrade proportionally to the estimated cost ratio, as the query with the lower cost does not rely as much on cache *whereas* the query with higher cost will revisit the same data over and over (triggering additional I/O in the absence of a stable cache), e.g.:

cost	time (dedicated machine)	time (under load)
-----	-----	-----
	+	+

```
some query A: 5k | (all data cached) 900ms | (less i/o) 1000ms |
some query B: 50k | (all data cached) 900ms | (lots of i/o) 10000ms |
```

Do not forget to run `ANALYZE lives` once after creating the necessary indices.

Query #1

```
-- incrementally narrow down the result set via inner joins
-- the CBO may elect to perform one full index scan combined
-- with cascading index lookups, or as hash aggregates terminated
-- by one nested index lookup into lives - on my machine
-- the latter query plan was selected given my memory settings and
-- histogram
SELECT
  l1.*
FROM
  lives AS l1
INNER JOIN (
  SELECT
    usr_id,
    MAX(time_stamp) AS time_stamp_max
  FROM
    lives
  GROUP BY
    usr_id
) AS l2
ON
  l1.usr_id = l2.usr_id AND
  l1.time_stamp = l2.time_stamp_max
INNER JOIN (
  SELECT
    usr_id,
    time_stamp,
    MAX(trans_id) AS trans_max
  FROM
    lives
  GROUP BY
    usr_id, time_stamp
) AS l3
ON
  l1.usr_id = l3.usr_id AND
  l1.time_stamp = l3.time_stamp AND
  l1.trans_id = l3.trans_max
```

Query #2

```
-- cheat to obtain a max of the (time_stamp, trans_id) tuple in one pass
-- this results in a single table scan and one nested index lookup into lives,
-- by far the least I/O intensive operation even in case of great scarcity
-- of memory (least reliant on cache for the best performance)
SELECT
  l1.*
FROM
  lives AS l1
INNER JOIN (
  SELECT
```

```

    usr_id,
    MAX(ARRAY[EXTRACT(EPOCH FROM time_stamp),trans_id])
      AS compound_time_stamp
FROM
  lives
GROUP BY
  usr_id
) AS l2
ON
  l1.usr_id = l2.usr_id AND
  EXTRACT(EPOCH FROM l1.time_stamp) = l2.compound_time_stamp[1] AND
  l1.trans_id = l2.compound_time_stamp[2]

```

2013/01/29 update

Finally, as of version 8.4, Postgres supports [Window Function](#) meaning you can write something as simple and efficient as:

Query #3

```

-- use Window Functions
-- performs a SINGLE scan of the table
SELECT DISTINCT ON (usr_id)
  last_value(time_stamp) OVER wnd,
  last_value(lives_remaining) OVER wnd,
  usr_id,
  last_value(trans_id) OVER wnd
FROM lives
WINDOW wnd AS (
  PARTITION BY usr_id ORDER BY time_stamp, trans_id
  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
);

```

Share Follow

edited Feb 2, 2016 at 13:31



dirkbaechle

4,052 ● 15 ● 18

answered Feb 26, 2009 at 1:19



vladr

66.6k ● 18 ● 130 ● 132

By a compound index on (usr_id, trans_id, times_tamp), do you mean something like "CREATE INDEX lives_blah_idx ON lives (usr_id, trans_id, time_stamp)"? Or should I create three separate indexes for each column? I should stick with the default of "USING btree", right? – [Joshua Berry](#) Feb 26, 2009 at 17:02

- 1 Yes to the first choice: I mean CREATE INDEX lives_blah_idx ON lives (usr_id, trans_id, time_stamp). :) Cheers. – [vladr](#) Feb 26, 2009 at 19:52

Thanks for even doing the cost comparison vladr! Very complete answer! – [Adam](#) Nov 30, 2010 at 0:34

@vladr I just came across your answer. I am a bit confused, as you say query 1 has a cost of 16k and query 2 a cost of 14k. But further down in the table you say query 1 has a cost of 5k and query 2 has a cost of 50k. So which query is a preferred one to use? :) thanks – [Houman](#) Jul 30, 2012 at 12:50

- 1 @Kave, the table is for a hypothetical pair of queries to illustrate an example, not the OP's two queries. Renaming to reduce confusion. – [vladr](#) Aug 13, 2012 at 20:15



15



There is a new option in Postgresql 9.5 called DISTINCT ON

```
SELECT DISTINCT ON (location) location, time, report
FROM weather_reports
ORDER BY location, time DESC;
```

It eliminates duplicate rows and leaves only the first row as defined by the ORDER BY clause.

see the official [documentation](#)

Share Follow

answered Jun 28, 2018 at 7:38



[Eden](#)

4,176 ● 3 ● 27 ● 26

Excellent, BTW specifying range of where that has index will greatly speed up by avoiding seq scan. – [Eric](#) Sep 18, 2021 at 11:37

Unfortunately this doesn't work if you are trying to get a distinct on a different column than you are ordering by. Such as when you have a table with a removed PK and duplicate values in the PK column.... – [Douglas Gaskell](#) Jan 30 at 21:24



11



Here's another method, which happens to use no correlated subqueries or GROUP BY. I'm not expert in PostgreSQL performance tuning, so I suggest you try both this and the solutions given by other folks to see which works better for you.

```
SELECT l1.*
FROM lives l1 LEFT OUTER JOIN lives l2
ON (l1.usr_id = l2.usr_id AND (l1.time_stamp < l2.time_stamp
OR (l1.time_stamp = l2.time_stamp AND l1.trans_id < l2.trans_id)))
WHERE l2.usr_id IS NULL
ORDER BY l1.usr_id;
```

I am assuming that `trans_id` is unique at least over any given value of `time_stamp`.

Share Follow

[edited Feb 25, 2009 at 18:37](#)

answered Feb 25, 2009 at 18:26



[Bill Karwin](#)

561k ● 87 ● 698 ● 854

- 1 It is cool that this is portable across RDMS, related: stackoverflow.com/questions/121387/...
– [Ciro Santilli OurBigBook.com](#) Jun 29, 2022 at 9:47



8



I like the style of [Mike Woodhouse's answer](#) on the other page you mentioned. It's especially concise when the thing being maximised over is just a single column, in which case the subquery can just use `MAX(some_col)` and `GROUP BY` the other columns, but in your case you have a 2-part quantity to be maximised, you can still do so by using `ORDER BY` plus `LIMIT 1` instead (as done by Quassnoi):

```
SELECT *
FROM lives outer
WHERE (usr_id, time_stamp, trans_id) IN (
    SELECT usr_id, time_stamp, trans_id
    FROM lives sq
    WHERE sq.usr_id = outer.usr_id
    ORDER BY trans_id, time_stamp
    LIMIT 1
)
```

I find using the row-constructor syntax `WHERE (a, b, c) IN (subquery)` nice because it cuts down on the amount of verbiage needed.

Share Follow

edited May 23, 2017 at 12:34



Community Bot

1 • 1

answered Jun 15, 2010 at 6:46



[j_random_hacker](#)

51.2k • 10 • 108 • 173



5



Actaually there's a hacky solution for this problem. Let's say you want to select the biggest tree of each forest in a region.

```
SELECT (array_agg(tree.id ORDER BY tree_size.size))[1]
FROM tree JOIN forest ON (tree.forest = forest.id)
GROUP BY forest.id
```

When you group trees by forests there will be an unsorted list of trees and you need to find the biggest one. First thing you should do is to sort the rows by their sizes and select the first one of your list. It may seems inefficient but if you have millions of rows it will be quite faster than the solutions that includes `JOIN`'s and `WHERE` conditions.

BTW, note that `ORDER_BY` for `array_agg` is introduced in Postgresql 9.0

Share Follow

edited Jul 12, 2015 at 10:05



Eric


97.5k • 54 • 254 • 385

answered Jan 18, 2013 at 0:04



[burak emre](#)

1,581 • 4 • 25 • 48

You have an error. You need to write ORDER BY tree_size.size DESC. Also, for author's task the code will look like this: `SELECT usr_id, (array_agg(time_stamp ORDER BY time_stamp DESC))[1] AS timestamp, (array_agg(lives_remaining ORDER BY time_stamp DESC))[1] AS lives_remaining, (array_agg(trans_id ORDER BY time_stamp DESC))[1] AS trans_id FROM lives GROUP BY usr_id` – alexkovelsky
Aug 21, 2014 at 14:45 

▲ You can do it with window functions

2

```
SELECT t.*
FROM
  (SELECT
    *,
    ROW_NUMBER() OVER(PARTITION BY usr_id ORDER BY time_stamp DESC) as r
   FROM lives) as t
WHERE t.r = 1
```

Share Follow

edited Dec 18, 2020 at 6:32

answered Dec 12, 2019 at 5:40



Turbcool

104 ● 8

- 1 I liked this method because 1) the exact same works on SQLite which [WONTFIXED DISTINCT on](#) (not sure if ANSI though) 2) if you change `ROW_NUMBER` with `RANK` you can get all rows in case of a tie. – [Ciro Santilli OurBigBook.com](#) Jul 15, 2022 at 15:32

```

SELECT l.*
FROM (
    SELECT DISTINCT usr_id
    FROM lives
) lo, lives l
WHERE l.ctid = (
    SELECT ctid
    FROM lives li
    WHERE li.usr_id = lo.usr_id
    ORDER BY
        time_stamp DESC, trans_id DESC
    LIMIT 1
)

```

Creating an index on `(usr_id, time_stamp, trans_id)` will greatly improve this query.

You should always, always have some kind of `PRIMARY KEY` in your tables.

Share Follow

answered Feb 25, 2009 at 17:01



Quassnoi

425k ● 93 ● 625 ● 619

This is a fun working solution:

```

with t (time_stamp, lives_remaining, usr_id, trans_id) as (
    values
        (time '07:00', 1, 1, 1),
        (time '09:00', 4, 2, 2),
        (time '10:00', 2, 3, 3),
        (time '10:00', 1, 2, 4),
        (time '11:00', 4, 1, 5),
        (time '11:00', 3, 1, 6),
        (time '13:00', 3, 3, 1)
)
select *
from unnest((
    select array_agg(r)
    from (
        select (max(array[row(time_stamp, lives_remaining, usr_id, trans_id)]))[1]
        r
        from t
        group by usr_id
    ) x
)) as x (time_stamp time, lives_remaining int, usr_id int, trans_id int)

```

It uses several tricks:

- `MAX(record)` isn't defined, but `MAX(array)` is, and the array can contain records
- The `UNNEST((SELECT ARRAY_AGG(...) ...))` trick allows for unnesting the nested record again

This produces:

time_stamp	lives_remaining	usr_id	trans_id
11:00:00	4	1	5
13:00:00	3	3	1
10:00:00	1	2	4

If you don't mind the nesting, then this is much simpler:

```
select (max(array[row(time_stamp, lives_remaining, usr_id, trans_id)]))[1] r
from t
group by usr_id
```

Producing:

r
(11:00:00, 4, 1, 5)
(13:00:00, 3, 3, 1)
(10:00:00, 1, 2, 4)

Finally, if `t` is an actual table, then you can cast the `row` to `t` in order to unnest the record more simply:

```
create table t (time_stamp time, lives_remaining int, usr_id int, trans_id
int);
insert into t
values
    (time '07:00', 1, 1, 1),
    (time '09:00', 4, 2, 2),
    (time '10:00', 2, 3, 3),
    (time '10:00', 1, 2, 4),
    (time '11:00', 4, 1, 5),
    (time '11:00', 3, 1, 6),
    (time '13:00', 3, 3, 1);

select ((max(array[row(time_stamp, lives_remaining, usr_id, trans_id)::t]))
[1]).*
from t
group by usr_id
```

Producing again:

time_stamp	lives_remaining	usr_id	trans_id
13:00:00	3	3	1
10:00:00	1	2	4
11:00:00	4	1	5

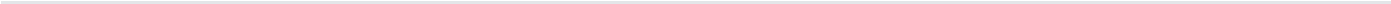
Share Follow

edited Jul 9, 2023 at 8:57

answered Jul 8, 2023 at 10:52



Lukas Eder
220k ● 135 ● 718 ● 1.6k





0



I think you've got one major problem here: there's no monotonically increasing "counter" to guarantee that a given row has happened later in time than another. Take this example:

timestamp	lives_remaining	user_id	trans_id
10:00	4	3	5
10:00	5	3	6
10:00	3	3	1
10:00	2	3	2

You cannot determine from this data which is the most recent entry. Is it the second one or the last one? There is no sort or max() function you can apply to any of this data to give you the correct answer.

Increasing the resolution of the timestamp would be a huge help. Since the database engine serializes requests, with sufficient resolution you can guarantee that no two timestamps will be the same.

Alternatively, use a trans_id that won't roll over for a very, very long time. Having a trans_id that rolls over means you can't tell (for the same timestamp) whether trans_id 6 is more recent than trans_id 1 unless you do some complicated math.

Share Follow

answered Feb 26, 2009 at 1:48



Barry Brown

20.6k ● 15 ● 71 ● 106

Yes, ideally a sequence (autoincrement) column would be in order. – [vladr](#) Feb 26, 2009 at 2:20

The assumption from above was that for small time increments, trans_id would not roll over. I agree that the table needs a unique primary index --like a non-repeating trans_id. (P.S. I'm happy that I now have enough karma/reputation points to comment!) – [Joshua Berry](#) Feb 26, 2009 at 11:55

Vlad states that trans_id has a rather short cycle which turns over frequently. Even if you consider only the middle two rows from my table (trans_id = 6 and 1), you still can't tell which is the most recent. Therefore, using the max(trans_id) for a given timestamp won't work. – [Barry Brown](#) Feb 26, 2009 at 18:32

Yep, I'm relying on the application author's guarantee that the (time_stamp,trans_id) tuple is unique for a given user. If it's not the case then "SELECT l1.usr_id,l1.lives_left,... FROM ... WHERE ..." must become "SELECT l1.usr_id,MAX/MIN(l1.lives_left),... FROM ... WHERE ... GROUP BY l1.usr_id,... – [vladr](#) Feb 26, 2009 at 19:56
