Select first row in each GROUP BY group?

Asked 14 years, 2 months ago Modified 2 months ago Viewed 1.9m times

2046

I'd like to select the first row of each set of rows grouped with a `GROUP BY`.

Specifically, if I've got a `purchases` table that looks like this:

```
SELECT * FROM purchases;
```

My Output:

id	customer	total
1	Joe	5
2	Sally	3
3	Joe	2
4	Sally	1

I'd like to query for the `id` of the largest purchase (`total`) made by each `customer`. Something like this:

```
SELECT FIRST(id), customer, FIRST(total)
FROM  purchases
GROUP BY customer
ORDER BY total DESC;
```

Expected Output:

FIRST(id)	customer	FIRST(total)
1	Joe	5
2	Sally	3

sql postgresql greatest-n-per-group

Share
Improve this question
Follow

edited Aug 22, 2023 at 23:44
Erwin Brandstetter
654k ● 156 ● 1.1k ● 1.3k

asked Sep 27, 2010 at 1:23
David Wolever
154k ● 93 ● 361 ● 509

1 since you are only looking for each largest one, why not query for `MAX(total)` ? – phil294 Oct 19, 2019 at 14:40

42 @phil294 querying for max(total) will not associate that total with the 'id' value of the row on which it occurred. – gwideman Feb 7, 2020 at 1:19

2 Does this answer your question? How do I select the first row per group in an SQL Query? – mafu Apr 23, 2021 at 17:58

Related or duplicate of stackoverflow.com/q/121387/6064933 – jdhao Sep 30, 2022 at 9:39

21 Answers

Sorted by: Highest score (default)

1828

+50

`DISTINCT ON` is typically simplest and fastest for this in **PostgreSQL**.

(For performance optimization for certain workloads see below.)

```
SELECT DISTINCT ON (customer)
       id, customer, total
FROM   purchases
ORDER  BY customer, total DESC, id;
```

Or shorter (if not as clear) with ordinal numbers of output columns:

```
SELECT DISTINCT ON (2)
       id, customer, total
FROM   purchases
ORDER  BY 2, 3 DESC, 1;
```

If `total` can be `null`, add `NULLS LAST`:

```
...
ORDER  BY customer, total DESC NULLS LAST, id;
```

Works either way, but you'll want to match existing indexes

db<>fiddle here

Major points

`DISTINCT ON` is a PostgreSQL extension of the standard, where only `DISTINCT` on the whole `SELECT` list is defined.

List any number of expressions in the `DISTINCT ON` clause, the combined row value defines duplicates. The manual:

> Obviously, two rows are considered distinct if they differ in at least one column value. **Null values are considered equal in this comparison.**

Bold emphasis mine.

`DISTINCT ON` can be combined with **`ORDER BY`**. Leading expressions in `ORDER BY` must be in the set of expressions in `DISTINCT ON`, but you can rearrange order among those freely. Example.
You can add *additional* expressions to `ORDER BY` to pick a particular row from each group of peers. Or, as the manual puts it:

> The `DISTINCT ON` expression(s) must match the leftmost `ORDER BY` expression(s). The `ORDER BY` clause will normally contain additional expression(s) that determine the desired precedence of rows within each `DISTINCT ON` group.

I added `id` as last item to break ties:
"Pick the row with the smallest `id` from each group sharing the highest `total`."

To order results in a way that disagrees with the sort order determining the first per group, you can nest above query in an outer query with another `ORDER BY`. Example.

If `total` can be `null`, you *most probably* want the row with the greatest non-null value. Add **`NULLS LAST`** like demonstrated. See:

- Sort by column ASC, but NULL values first?

The `SELECT` list is not constrained by expressions in `DISTINCT ON` or `ORDER BY` in any way:

- You *don't have to* include any of the expressions in `DISTINCT ON` or `ORDER BY`.
- You *can* include any other expression in the `SELECT` list. This is instrumental for replacing complex subqueries and aggregate / window functions.

I tested with Postgres versions 8.3 – 17. But the feature has been there at least since version 7.1, so basically always.

Index

The *perfect* index for the above query would be a multi-column index spanning all three columns in matching sequence and with matching sort order:

```
CREATE INDEX purchases_3c_idx ON purchases (customer, total DESC, id);
```

May be too specialized. But use it if read performance for the particular query is crucial. If you have `DESC NULLS LAST` in the query, use the same in the index so that sort order matches and the index is perfectly applicable.

Effectiveness / Performance optimization

Weigh cost and benefit before creating tailored indexes for each query. The potential of above index largely depends on **data distribution**.

The index is used because it delivers pre-sorted data. In Postgres 9.2 or later the query can also benefit from an **index only scan** if the index is smaller than the underlying table. The index has to be scanned in its entirety, though. Example.

For ***few*** **rows per customer** (high cardinality in column `customer`), this is very efficient. Even more so if you need sorted output anyway. The benefit shrinks with a growing number of rows per customer.
Ideally, you have enough **`work_mem`** to process the involved sort step in RAM and not spill to disk. But generally setting `work_mem` *too* high can have adverse effects. Consider `SET LOCAL` for exceptionally big queries. Find how much you need with `EXPLAIN ANALYZE`. Mention of "*Disk:*" in the sort step indicates the need for more:

- Configuration parameter work_mem in PostgreSQL on Linux
- Optimize simple query using ORDER BY date and text

For ***many*** **rows per customer** (low cardinality in column `customer`), an "**index skip scan**" or **"loose index scan"** would be (much) more efficient. But that's not implemented up to Postgres 17. Serious work to implement it one way or another has been ongoing for years now, but so far unsuccessful. See here and here.
For now, there are **faster query techniques** to substitute for this. In particular if you have a separate table holding unique customers, which is the typical use case. But also if you don't:

- **Optimize GROUP BY query to retrieve latest row per user**
- SELECT DISTINCT is slower than expected on my table in PostgreSQL
- Optimize groupwise maximum query
- Query last N related rows per row

Benchmarks

See separate answer.

Share

Improve this answer

Follow

edited Oct 16 at 22:41

answered Oct 3, 2011 at 2:21

Erwin Brandstetter
654k ● 156 ● 1.1k ● 1.3k

sadly if you want to order and distinguish by different logics, DISTINCT ON is useless and you have to use a subquery – zoltankundi Jan 12, 2023 at 11:47

@zoltankundi: Why would a subquery make `DISTINCT ON` useless? I guess it's about cases like this one? stackoverflow.com/a/9796104/939860 – Erwin Brandstetter Jan 14, 2023 at 6:38

1 I'm not saying a subquery makes it useless, just that you have to use a subquery and it would be nice if you could do DISTINCT ON without having to also sort by the same column – zoltankundi Jan 15, 2023 at 16:12

2 You can use `DISTINCT ON` without `ORDER BY`. Just not with a contradicting `ORDER BY`. For that you need a subquery. – Erwin Brandstetter Jan 15, 2023 at 16:18

1510

On databases that support CTE and windowing functions:

```
WITH summary AS (
    SELECT p.id,
           p.customer,
           p.total,
           ROW_NUMBER() OVER(PARTITION BY p.customer
                                 ORDER BY p.total DESC) AS rank
      FROM PURCHASES p)
 SELECT *
   FROM summary
 WHERE rank = 1
```

Supported by any database:

But you need to add logic to break ties:

```
  SELECT MIN(x.id),  -- change to MAX if you want the highest
         x.customer,
         x.total
    FROM PURCHASES x
    JOIN (SELECT p.customer,
                 MAX(total) AS max_total
            FROM PURCHASES p
        GROUP BY p.customer) y ON y.customer = x.customer
                              AND y.max_total = x.total
GROUP BY x.customer, x.total
```

Share

Improve this answer

Follow

edited May 6, 2021 at 17:15
rogerdpack
66.5k ● 39 ● 282 ● 401

answered Sep 27, 2010 at 1:27
OMG Ponies
332k ● 85 ● 534 ● 507

73 `ROW_NUMBER() OVER(PARTITION BY [...])` along with some other optimizations helped me get a query down from 30 seconds to a few milliseconds. Thanks! (PostgreSQL 9.2) – Sam Oct 1, 2014 at 21:29

2 ROW_NUMBER() OVER(PARTITION meets my needs but Is there any way to limit the row numbers to just 1 from the group to reduce the size of the view? – Solomon Tesfaye Jul 13, 2022 at 10:43

1 @SolomonTesfaye Use a subquery and in the view specify `WHERE row_number = 1` against the subquery. – hemp Jun 20, 2023 at 0:46

1 @hemp The summary view has already big data and I have to filter from the result using `where rank=1`. But I was asking if there are ways to reduce the view at the first place. – Solomon Tesfaye Jun 20, 2023 at 1:32

@SolomonTesfaye You can use a different approach, such as DISTINCT ON instead. – hemp Jun 20, 2023 at 6:10

296

Benchmarks

I tested the most interesting candidates:

- Initially with **Postgres 9.4** and **9.5**.
- Added accented tests for **Postgres 13** later.

Basic test setup

Main table: `purchases`:

```
CREATE TABLE purchases (
  id          serial   -- PK constraint added below
, customer_id int      -- REFERENCES customer
, total       int      -- could be amount of money in Cent
, some_column text     -- to make the row bigger, more realistic
);
```

Dummy data (with some dead tuples), PK, index:

```
INSERT INTO purchases (customer_id, total, some_column)    -- 200k rows
SELECT (random() * 10000)::int             AS customer_id  -- 10k distinct
customers
     , (random() * random() * 100000)::int AS total
     , 'note: ' || repeat('x', (random()^2 * random() * random() * 500)::int)
FROM   generate_series(1,200000) g;
```

```
ALTER TABLE purchases ADD CONSTRAINT purchases_id_pkey PRIMARY KEY (id);

DELETE FROM purchases WHERE random() > 0.9;  -- some dead rows

INSERT INTO purchases (customer_id, total, some_column)
SELECT (random() * 10000)::int             AS customer_id  -- 10k customers
     , (random() * random() * 100000)::int AS total
     , 'note: ' || repeat('x', (random()^2 * random() * random() * 500)::int)
FROM   generate_series(1,20000) g;  -- add 20k to make it ~ 200k

CREATE INDEX purchases_3c_idx ON purchases (customer_id, total DESC, id);

VACUUM ANALYZE purchases;
```

`customer` table - used for optimized query:

```
CREATE TABLE customer AS
SELECT customer_id, 'customer_' || customer_id AS customer
FROM   purchases
GROUP  BY 1
ORDER  BY 1;

ALTER TABLE customer ADD CONSTRAINT customer_customer_id_pkey PRIMARY KEY
(customer_id);

VACUUM ANALYZE customer;
```

In my second test for 9.5 I used the same setup, but with 100000 distinct `customer_id` to get *few* rows per `customer_id`.

Object sizes for table `purchases`

Basic setup: 200k rows in `purchases`, 10k distinct `customer_id`, avg. 20 rows per customer.
For Postgres 9.5 I added a 2nd test with 86446 distinct customers - avg. 2.3 rows per customer.

Generated with a query taken from here:

- Measure the size of a PostgreSQL table row

Gathered for Postgres 9.5:

```
               what                | bytes/ct | bytes_pretty | bytes_per_row
-----------------------------------+----------+--------------+---------------
 core_relation_size                | 20496384 | 20 MB        |           102
 visibility_map                    |        0 | 0 bytes      |             0
 free_space_map                    |    24576 | 24 kB        |             0
 table_size_incl_toast             | 20529152 | 20 MB        |           102
 indexes_size                      | 10977280 | 10 MB        |            54
 total_size_incl_toast_and_indexes | 31506432 | 30 MB        |           157
 live_rows_in_text_representation    | 13729802 | 13 MB          |          68
 ------------------------------      |          |                |
 row_count                           |   200045 |                |
 live_tuples                         |   200045 |                |
 dead_tuples                         |    19955 |                |
```

Queries

1. `row_number()` in CTE, (see other answer)

```
WITH cte AS (
   SELECT id, customer_id, total
        , row_number() OVER (PARTITION BY customer_id ORDER BY total DESC) AS
rn
   FROM   purchases
   )
SELECT id, customer_id, total
FROM   cte
WHERE  rn = 1;
```

2. `row_number()` in subquery (my optimization)

```
SELECT id, customer_id, total
FROM   (
   SELECT id, customer_id, total
        , row_number() OVER (PARTITION BY customer_id ORDER BY total DESC) AS
rn
   FROM   purchases
   ) sub
WHERE  rn = 1;
```

3. `DISTINCT ON` (see other answer)

```
SELECT DISTINCT ON (customer_id)
       id, customer_id, total
FROM   purchases
ORDER  BY customer_id, total DESC, id;
```

4. rCTE with `LATERAL` subquery (see here)

```
WITH RECURSIVE cte AS (
   (  -- parentheses required
   SELECT id, customer_id, total
   FROM   purchases
   ORDER  BY customer_id, total DESC
   LIMIT  1
   )
   UNION ALL
   SELECT u.*
   FROM   cte c
   ,      LATERAL (
      SELECT id, customer_id, total
      FROM   purchases
      WHERE  customer_id > c.customer_id  -- lateral reference
      ORDER  BY customer_id, total DESC
      LIMIT  1
      ) u
   )
SELECT id, customer_id, total
FROM   cte
ORDER  BY customer_id;
```

5. `customer` table with `LATERAL` (see here)

```
SELECT l.*
FROM   customer c
,      LATERAL (
   SELECT id, customer_id, total
   FROM   purchases
   WHERE  customer_id = c.customer_id  -- lateral reference
   ORDER  BY total DESC
   LIMIT  1
   ) l;
```

6. `array_agg()` with `ORDER BY` (see other answer)

```
SELECT (array_agg(id ORDER BY total DESC))[1] AS id
     , customer_id
     , max(total) AS total
FROM   purchases
GROUP  BY customer_id;
```

Results

Execution time for above queries with `EXPLAIN (ANALYZE, TIMING OFF, COSTS OFF`, *best of 5 runs* to compare with warm cache.

All queries used an **Index Only Scan** on `purchases2_3c_idx` (among other steps). Some only to benefit from the smaller size of the index, others more effectively.

A. Postgres 9.4 with 200k rows and ~ 20 per `customer_id`

```
1. 273.274 ms
2. 194.572 ms
3. 111.067 ms
4.  92.922 ms  -- !
```

```
5.  37.679 ms  -- winner
6. 189.495 ms
```

B. Same as A. with Postgres 9.5

```
1. 288.006 ms
2. 223.032 ms
3. 107.074 ms
4.  78.032 ms  -- !
5.  33.944 ms  -- winner
6. 211.540 ms
```

C. Same as B., but with ~ 2.3 rows per `customer_id`

```
1. 381.573 ms
2. 311.976 ms
3. 124.074 ms  -- winner
4. 710.631 ms
5. 311.976 ms
6. 421.679 ms
```

Retest with Postgres 13 on 2021-08-11

Simplified test setup: no deleted rows, because `VACUUM ANALYZE` cleans the table completely for the simple case.

Important changes for Postgres:

- General performance improvements.
- CTEs can be inlined since Postgres 12, so query 1. and 2. now perform mostly identical (same query plan).

D. Like B. ~ 20 rows per customer_id

```
1. 103 ms
2. 103 ms
3.  23 ms  -- winner
4.  71 ms
5.  22 ms  -- winner
6.  81 ms
```

db<>fiddle [here]

E. Like C. ~ 2.3 rows per customer_id

```
1. 127 ms
2. 126 ms
3.  36 ms  -- winner
4. 620 ms
5. 145 ms
6. 203 ms
```

db<>fiddle [here](#)

Accented tests with Postgres 13

1M rows, 10.000 vs. 100 vs. 1.6 rows per customer.

F. with ~ 10.000 rows per customer

```
1. 526 ms
2. 527 ms
3. 127 ms
4.   2 ms  -- winner !
5.   1 ms  -- winner !
6. 356 ms
```

db<>fiddle [here](#)

G. with ~ 100 rows per customer

```
1. 535 ms
2. 529 ms
3. 132 ms
4. 108 ms  -- !
5.  71 ms  -- winner
6. 376 ms
```

db<>fiddle [here](#)

H. with ~ 1.6 rows per customer

```
1.  691 ms
2.  684 ms
3.  234 ms  -- winner
4. 4669 ms
5. 1089 ms
6. 1264 ms
```

db<>fiddle [here](#)

Conclusions

- `DISTINCT ON` uses the index effectively and typically performs best for **few** rows per group. And it performs decently even with many rows per group.
- For **many** rows per group, emulating an index skip scan with an rCTE performs best - second only to the query technique with a separate lookup table (if that's available).
- The **`row_number()`** technique demonstrated in the currently accepted answer **never wins any performance test**. Not then, not now. It never comes even close to `DISTINCT ON`, not even when the data distribution is unfavorable for the latter. The only good thing about `row_number()`: it does not scale terribly, just mediocre.

More benchmarks

Benchmark by "ogr" with **10M rows and 60k unique "customers"** on **Postgres 11.5**. Results are in line with what we have seen so far:

- Proper way to access latest row for each individual identifier?

Original (outdated) benchmark from 2011

I ran three tests with PostgreSQL **9.1** on a real life table of 65579 rows and single-column btree indexes on each of the three columns involved and took the best *execution time* of 5 runs.
Comparing @OMGPonies' first query (**`A`**) to the above `DISTINCT ON` solution (**`B`**):

1. Select the whole table, results in 5958 rows in this case.

```
A: 567.218 ms
B: 386.673 ms
```

2. Use condition `WHERE customer BETWEEN x AND y` resulting in 1000 rows.

```
A: 249.136 ms
B:  55.111 ms
```

3. Select a single customer with `WHERE customer = x`.

```
A:   0.143 ms
B:   0.072 ms
```

Same test repeated with the index described in the other answer:

```
CREATE INDEX purchases_3c_idx ON purchases (customer, total DESC, id);
```

```
1A: 277.953 ms
1B: 193.547 ms

2A: 249.796 ms -- special index not used
2B:  28.679 ms

3A:   0.120 ms
3B:   0.048 ms
```

Share

Improve this answer

Follow

edited May 22, 2022 at 21:25

answered Jan 11, 2016 at 6:05

Erwin Brandstetter
654k ● 156 ● 1.1k ● 1.3k

1 can please you add custom aggregate method to the benchmark? something like "select first(purchases order by id) from purchases group by customer" wiki.postgresql.org/wiki/First/last_(aggregate) – Adithya Sama Nov 2, 2021 at 9:02

Yes, we can please add the group by query in the benchmark. I know it's not exactly the same in case of duplicates. But it might be in a lot of use cases (like timestamp) and it's the first solution that folks think of: SELECT id, customer_id, total FROM purchases a JOIN (SELECT customer_id, MAX(total) AS total GROUP BY customer_id) b ON a.customer_id = b.customer AND a.total = b.total – na_ka_na Jan 31, 2022 at 22:51

72

This is common `greatest-n-per-group` problem, which already has well tested and highly optimized solutions. Personally I prefer the left join solution by Bill Karwin (the original post with lots of other solutions).

Note that bunch of solutions to this common problem can surprisingly be found in the **MySQL manual** -- even though your problem is in Postgres, not MySQL, the solutions given should work with most SQL variants. See Examples of Common Queries :: The Rows Holding the Group-wise Maximum of a Certain Column.

Share

Improve this answer

Follow

edited Sep 13, 2022 at 20:16

Ross Presser
6,245 ● 2 ● 37 ● 71

answered Jun 27, 2013 at 8:38

Tomas
59.4k ● 54 ● 248 ● 381

33 How is the MySQL manual in any way "official" for Postgres / SQLite (not to mention SQL) questions? Also, to be clear, the `DISTINCT ON` version is much shorter, simpler and generally performs better in Postgres than alternatives with a self `LEFT JOIN` or semi-anti-join with `NOT EXISTS`. It is also "well tested". – Erwin Brandstetter Jul 8, 2013 at 18:27

As commented under the mentioned "left join" solution, beware that self-joins cause performance quadratic in the group sizes, making them unsuitable for when groups can be large. Refer to the comment about self-joins under that answer for more details. – Timo Aug 14, 2023 at 17:12

48

In Postgres you can use `array_agg` like this:

```
SELECT  customer,
        (array_agg(id ORDER BY total DESC))[1],
        max(total)
FROM purchases
GROUP BY customer
```

This will give you the `id` of each customer's largest purchase.

Some things to note:

- `array_agg` is an aggregate function, so it works with `GROUP BY`.
- `array_agg` lets you specify an ordering scoped to just itself, so it doesn't constrain the structure of the whole query. There is also syntax for how you sort NULLs, if you need to do something different from the default.
- Once we build the array, we take the first element. (Postgres arrays are 1-indexed, not 0-indexed).
- You could use `array_agg` in a similar way for your third output column, but `max(total)` is simpler.
- Unlike `DISTINCT ON`, using `array_agg` lets you keep your `GROUP BY`, in case you want that for other reasons.

Share
Improve this answer
Follow

edited Aug 27, 2014 at 18:57

answered Aug 27, 2014 at 18:14
Paul A Jungwirth
24.5k ● 15 ● 78 ● 94

20

The Query:

```
SELECT purchases.*
FROM purchases
LEFT JOIN purchases as p
ON
  p.customer = purchases.customer
  AND
  purchases.total < p.total
WHERE p.total IS NULL
```

HOW DOES THAT WORK! (I've been there)

We want to make sure that we only have the highest total for each purchase.

Some Theoretical Stuff (skip this part if you only want to understand the query)

Let Total be a function T(customer,id) where it returns a value given the name and id To prove that the given total (T(customer,id)) is the highest we have to prove that We want to prove either

- ∀x T(customer,id) > T(customer,x) (this total is higher than all other total for that customer)

OR

- ¬∃x T(customer, id) < T(customer, x) (there exists no higher total for that customer)

The first approach will need us to get all the records for that name which I do not really like.

The second one will need a smart way to say there can be no record higher than this one.

Back to SQL

If we left joins the table on the name and total being less than the joined table:

```
LEFT JOIN purchases as p
ON
p.customer = purchases.customer
AND
purchases.total < p.total
```

we make sure that all records that have another record with the higher total for the same user to be joined:

```
+--------------+---------------------+-----------------+------+------------+---
------+
| purchases.id |  purchases.customer | purchases.total | p.id | p.customer |
p.total |
+--------------+---------------------+-----------------+------+------------+---
------+
|            1 | Tom                 |             200 |    2 | Tom        |
300 |
|            2 | Tom                 |             300 |      |            |
|
|            3 | Bob                 |             400 |    4 | Bob        |
500 |
|            4 | Bob                 |             500 |      |            |
|
|            5 | Alice               |              600 |    6 | Alice       |
700 |
|            6 | Alice               |              700 |      |             |
|
+-------------+--------------------+----------------+------+-----------+---
------+
```

That will help us filter for the highest total for each purchase with no grouping needed:

```
WHERE p.total IS NULL

+-------------+---------------+----------------+------+--------+---------+
| purchases.id | purchases.name | purchases.total | p.id | p.name | p.total |
+-------------+---------------+----------------+------+--------+---------+
|            2 | Tom            |             300 |      |        |         |
|            4 | Bob            |             500 |      |        |         |
|            6 | Alice          |             700 |      |        |         |
+-------------+---------------+----------------+------+--------+---------+
```

And that's the answer we need.

Share

Improve this answer

Follow

edited Aug 1, 2020 at 21:12

Bitswazsky
4,678 ● 3 ● 36 ● 64

answered Mar 24, 2018 at 16:11

khaled_gomaa
3,412 ● 22 ● 24

1 Very neat solution. I'm curious how performant it comparing to others. Even if it is not the best, it is still interesting question. In current version of MariaDB I have not LATERAL, DISTINCT ON and ARRAY_AGG(), thus I have choice only between this solution and ROW_NUMBER() – kdmitry Jul 17, 2023 at 17:14

1 I did some testing and seems to be ROW_NUMBER() solution has better performance in my case. In short: 5 million of records, this solution - 5 m 24 s, ROW_NUMBER() solution - 1 m 40 s. After adding of index the difference even greater: this solution - 59.5 s, ROW_NUMBER() solution - 9.5 s. Remember, that your mileage may vary – kdmitry Aug 27, 2023 at 16:15

18

The solution is not very efficient as pointed by Erwin, because of presence of SubQs

```
select * from purchases p1 where total in
(select max(total) from purchases where p1.customer=customer) order by total
desc;
```

Share

Improve this answer

Follow

edited Jun 17, 2013 at 20:39

answered Jun 17, 2013 at 18:02

cosmos
2,303 ● 1 ● 18 ● 32

13

I use this way (postgresql only):
https://wiki.postgresql.org/wiki/First/last_%28aggregate%29

```
-- Create a function that always returns the first non-NULL item
CREATE OR REPLACE FUNCTION public.first_agg ( anyelement, anyelement )
RETURNS anyelement LANGUAGE sql IMMUTABLE STRICT AS $$
        SELECT $1;
$$;

-- And then wrap an aggregate around it
CREATE AGGREGATE public.first (
        sfunc    = public.first_agg,
        basetype = anyelement,
        stype    = anyelement
);

-- Create a function that always returns the last non-NULL item
CREATE OR REPLACE FUNCTION public.last_agg ( anyelement, anyelement )
RETURNS anyelement LANGUAGE sql IMMUTABLE STRICT AS $$
        SELECT $2;
$$;

-- And then wrap an aggregate around it
CREATE AGGREGATE public.last (
        sfunc    = public.last_agg,
        basetype = anyelement,
        stype    = anyelement
);
```

Then your example should work *almost* as is:

```
SELECT FIRST(id), customer, FIRST(total)
FROM  purchases
GROUP BY customer
ORDER BY FIRST(total) DESC;
```

CAVEAT: It ignore's NULL rows

Edit 1 - Use the postgres extension instead

Now I use this way: http://pgxn.org/dist/first_last_agg/

To install on ubuntu 14.04:

```
apt-get install postgresql-server-dev-9.3 git build-essential -y
git clone git://github.com/wulczer/first_last_agg.git
cd first_last_app
make && sudo make install
psql -c 'create extension first_last_agg'
```

It's a postgres extension that gives you first and last functions; apparently faster than the above way.

Edit 2 - Ordering and filtering

If you use aggregate functions (like these), you can order the results, without the need to have the data already ordered:

```
http://www.postgresql.org/docs/current/static/sql-expressions.html#SYNTAX-
AGGREGATES
```

So the equivalent example, with ordering would be something like:

```
SELECT first(id order by id), customer, first(total order by id)
  FROM purchases
 GROUP BY customer
 ORDER BY first(total);
```

Of course you can order and filter as you deem fit within the aggregate; it's very powerful syntax.

Share

Improve this answer

Follow

edited Mar 10, 2015 at 22:55

answered Mar 10, 2015 at 15:19

matiu

7,695 ● 5 ● 46 ● 49

12

Very fast solution

```
SELECT a.*
FROM
    purchases a
    JOIN (
        SELECT customer, min( id ) as id
        FROM purchases
        GROUP BY customer
    ) b USING ( id );
```

and really very fast if table is indexed by id:

```
create index purchases_id on purchases (id);
```

Share

Improve this answer

Follow

edited Aug 2, 2016 at 21:15

answered Apr 8, 2014 at 16:13

Alejandro Salamanca Mazuelo

1,351 ● 17 ● 24

12

Use `ARRAY_AGG` function for PostgreSQL, U-SQL, IBM DB2, and Google BigQuery SQL:

```
SELECT customer, (ARRAY_AGG(id ORDER BY total DESC))[1], MAX(total)
FROM purchases
GROUP BY customer
```

Share
Improve this answer
Follow

edited Apr 4, 2019 at 21:03

answered Apr 4, 2019 at 20:54
Valentin Podkamennyi
7,351 ● 4 ● 31 ● 44

11

In SQL Server you can do this:

```
SELECT *
FROM (
SELECT ROW_NUMBER()
OVER(PARTITION BY customer
ORDER BY total DESC) AS StRank, *
FROM Purchases) n
WHERE StRank = 1
```

Explaination:Here **Group by** is done on the basis of customer and then order it by total then each such group is given serial number as StRank and we are taking out first 1 customer whose StRank is 1

Share Improve this answer Follow

answered Dec 29, 2018 at 16:12
Diwas Poudel
847 ● 1 ● 11 ● 20

7

In PostgreSQL, another possibility is to use the `first_value` window function in combination with `SELECT DISTINCT`:

```
select distinct customer_id,
                first_value(row(id, total)) over(partition by customer_id order
by total desc, id)
from            purchases;
```

I created a composite `(id, total)`, so both values are returned by the same aggregate. You can of course always apply `first_value()` twice.

Share Improve this answer Follow

answered Dec 9, 2019 at 12:49

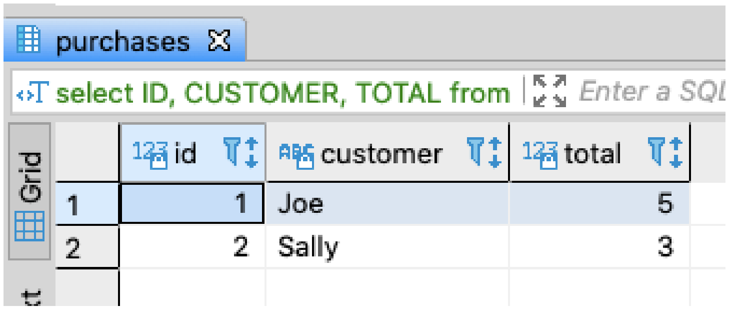
user8870331

7

This is how we can achieve this by using windows function:

```
create table purchases (id int4, customer varchar(10), total integer);
insert into purchases values (1, 'Joe', 5);
insert into purchases values (2, 'Sally', 3);
insert into purchases values (3, 'Joe', 2);
insert into purchases values (4, 'Sally', 1);

select ID, CUSTOMER, TOTAL from (
select ID, CUSTOMER, TOTAL,
row_number () over (partition by CUSTOMER order by TOTAL desc) RN
from purchases) A where RN = 1;
```



select ID, CUSTOMER, TOTAL from

	id	customer	total
1	1	Joe	5
2	2	Sally	3

Share Improve this answer Follow

answered Feb 7, 2022 at 4:30

PraveenP
796 ● 4 ● 7

6

Snowflake/Teradata supports `QUALIFY` clause which works like `HAVING` for windowed functions:

```
SELECT id, customer, total
FROM PURCHASES
QUALIFY ROW_NUMBER() OVER(PARTITION BY p.customer ORDER BY p.total DESC) = 1
```

Share Improve this answer Follow

answered Nov 17, 2019 at 21:19

Lukasz Szozda
175k ● 25 ● 267 ● 310

6

This way it work for me:

```
SELECT article, dealer, price
FROM   shop s1
WHERE  price=(SELECT MAX(s2.price)
              FROM shop s2
              WHERE s1.article = s2.article
```

```
                GROUP BY s2.article)
ORDER BY article;
```

Select highest price on each article

Share Improve this answer Follow

answered Jul 17, 2020 at 3:40

uncle bob
680 ● 1 ● 13 ● 24

5

The accepted OMG Ponies' "Supported by any database" solution has good speed from my test.

Here I provide a same-approach, but more complete and clean any-database solution. Ties are considered (assume desire to get only one row for each customer, even multiple records for max total per customer), and other purchase fields (e.g. purchase_payment_id) will be selected for the real matching rows in the purchase table.

Supported by any database:

```
select * from purchase
join (
    select min(id) as id from purchase
    join (
        select customer, max(total) as total from purchase
        group by customer
    ) t1 using (customer, total)
    group by customer
) t2 using (id)
order by customer
```

This query is reasonably fast especially when there is a composite index like (customer, total) on the purchase table.

Remark:

1. t1, t2 are subquery alias which could be removed depending on database.
2. **Caveat**: the `using (...)` clause is currently not supported in MS-SQL and Oracle db as of this edit on Jan 2017. You have to expand it yourself to e.g. `on t2.id = purchase.id` etc. The USING syntax works in SQLite, MySQL and PostgreSQL.

Share

Improve this answer

Follow

edited Jan 11, 2017 at 10:03

answered Jan 4, 2017 at 15:47

Johnny Wong
975 ● 9 ● 16

3

- If you want to select any (by your some specific condition) row from the set of aggregated rows.
- If you want to use another (`sum/avg`) aggregation function in addition to `max/min`. Thus you can not use clue with `DISTINCT ON`

You can use next subquery:

```
SELECT
    (
       SELECT **id** FROM t2
       WHERE id = ANY ( ARRAY_AGG( tf.id ) ) AND amount = MAX( tf.amount )
    ) id,
    name,
    MAX(amount) ma,
    SUM( ratio )
FROM t2  tf
GROUP BY name
```

You can replace `amount = MAX( tf.amount )` with any condition you want with one restriction: This subquery must not return more than one row

But if you wanna to do such things you probably looking for window functions

Share

Improve this answer

Follow

edited Sep 28, 2018 at 14:06

answered Sep 28, 2018 at 13:50

Eugen Konkov

25k ● 17 ● 122 ● 179

3

For SQl Server the most efficient way is:

```
with
ids as ( --condition for split table into groups
    select i from (values (9),(12),(17),(18),(19),(20),(22),(21),(23),(10)) as
v(i)
)
,src as (
    select * from yourTable where  <condition> --use this as filter for other
conditions
)
,joined as (
    select tops.* from ids
    cross apply --it`s like for each rows
    (
        select top(1) *
        from src
        where CommodityId = ids.i
    ) as tops
)
select * from joined
```

and don't forget to create clustered index for used columns

Share

Improve this answer

Follow

edited Jan 18, 2019 at 11:45

answered Jan 18, 2019 at 10:59

BazSTR
137 ● 4

1

This can be achieved easily by MAX FUNCTION on total and GROUP BY id and customer.

```
SELECT id, customer, MAX(total) FROM  purchases GROUP BY id, customer
ORDER BY total DESC;
```

Share

Improve this answer

Follow

edited Sep 9, 2022 at 11:59

Zoe - Save the data dump ♦
28.1k ● 22 ● 127 ● 158

answered Dec 16, 2021 at 9:43

Salman Sabir
69 ● 8

2 This doesn't do what the OP asks for. – Erwin Brandstetter Dec 16, 2021 at 22:55

2 If we know that the group always contains the same values or if we don't care which one to pick from the group, why not? In many cases, this is the best solution (only "order by" is not needed) – Sergey Shcherbakov Jul 25, 2022 at 16:21

1 "or if we don't care which one to pick from the group" but we DO care, hence the question. – bfontaine Aug 29, 2023 at 15:20

0

My approach via window function dbfiddle:

1. Assign `row_number` at each group: `row_number() over (partition by agreement_id, order_id ) as nrow`
2. Take only first row at group: `filter (where nrow = 1)`

```
with intermediate as (select
 *,
 row_number() over ( partition by agreement_id, order_id ) as nrow,
 (sum( suma ) over ( partition by agreement_id, order_id ))::numeric( 10, 2) as
order_suma,
from <your table>)

select
  *,
  sum( order_suma ) filter (where nrow = 1) over (partition by agreement_id)
from intermediate
```

Share Improve this answer Follow

answered May 13, 2021 at 13:18

Eugen Konkov
25k ● 17 ● 122 ● 179

0

you can get the first row in each group by using CTE (common table expression), below is the sample example

```
with cte as (SELECT t1.*
FROM table_one t1
INNER JOIN (
    SELECT id,MAX(date) AS max_date
    FROM table1
    GROUP BY id
) t2 ON t1.id = t2.id AND t1.max_date= t2.date)
```

Thanks

Share
Improve this answer
Follow

edited Dec 6, 2023 at 11:51
eng
789 ● 2 ● 6 ● 14

answered Sep 15, 2023 at 7:01
Programmer
75 ● 10

1 As it’s currently written, your answer is unclear. Please edit to add additional details that will help others understand how this addresses the question asked. You can find more information on how to write good answers in the help center. – Community Bot Sep 18, 2023 at 17:25