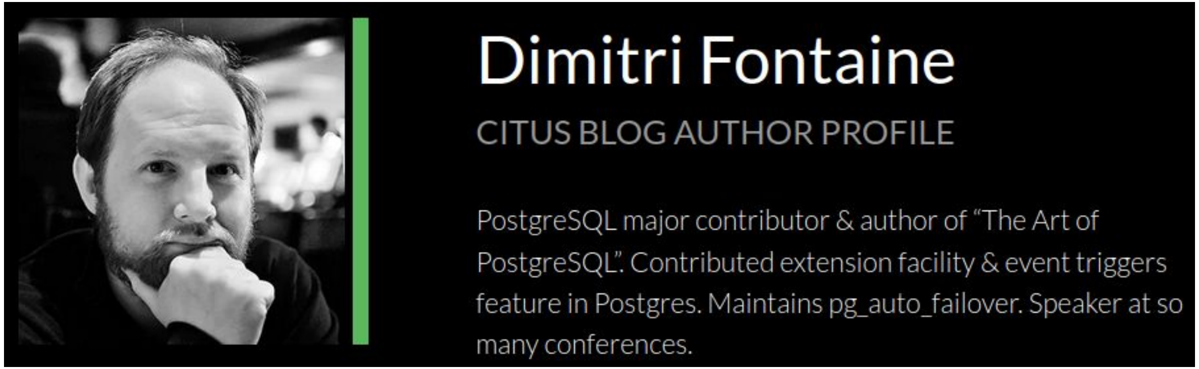
Statistics for Cost Estimation

Invited Speakers

Extensible Databases - 29/10/2021

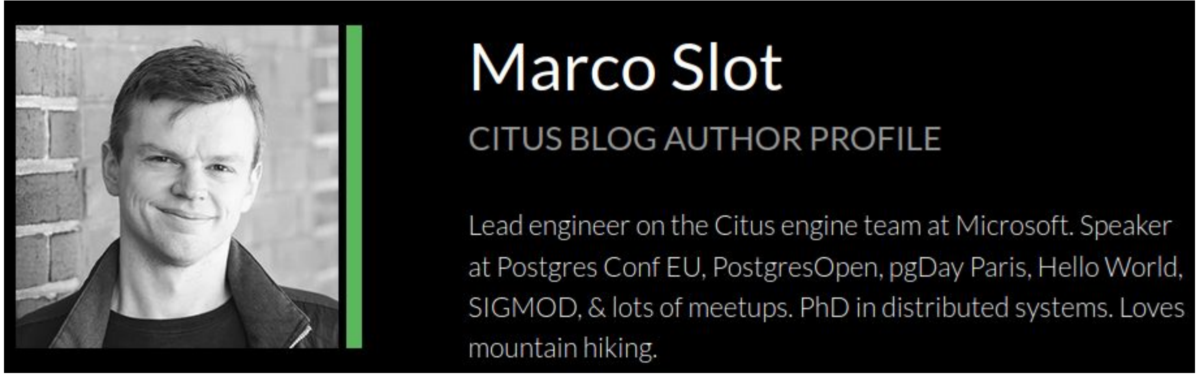


Dimitri Fontaine

CITUS BLOG AUTHOR PROFILE

PostgreSQL major contributor & author of "The Art of PostgreSQL". Contributed extension facility & event triggers feature in Postgres. Maintains pg_auto_failover. Speaker at so many conferences.

Distributed Databases - 26/11/2021



Marco Slot

CITUS BLOG AUTHOR PROFILE

Lead engineer on the Citus engine team at Microsoft. Speaker at Postgres Conf EU, PostgresOpen, pgDay Paris, Hello World, SIGMOD, & lots of meetups. PhD in distributed systems. Loves mountain hiking.

Statistical Information for Cost Estimation

- n_r: number of tuples in a relation *r.*
- b_r: number of blocks containing tuples of *r.*
- l_r: size of a tuple of *r.*
- f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
- *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\prod_A(r)$.
- If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Postgres Statistics

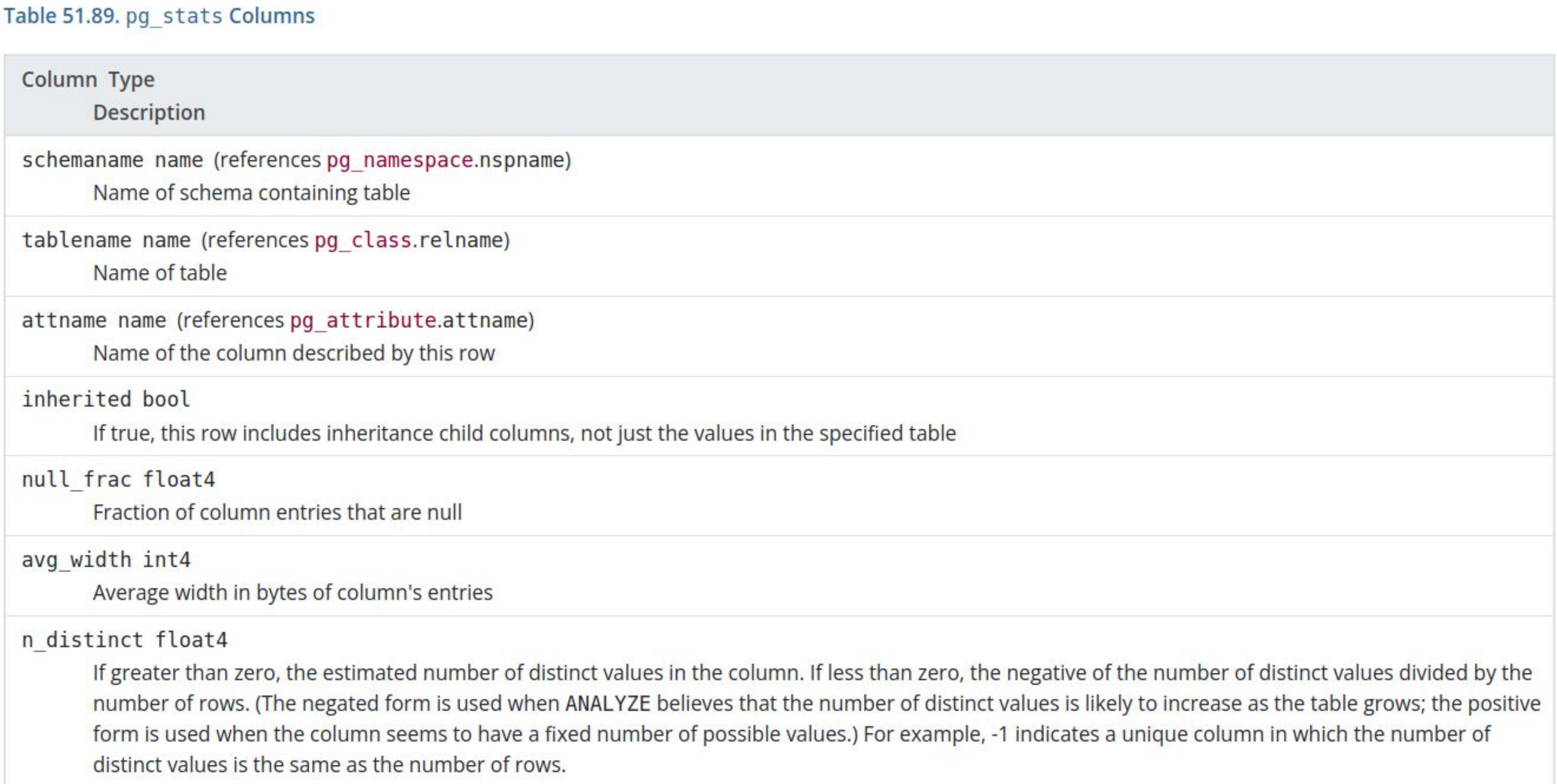
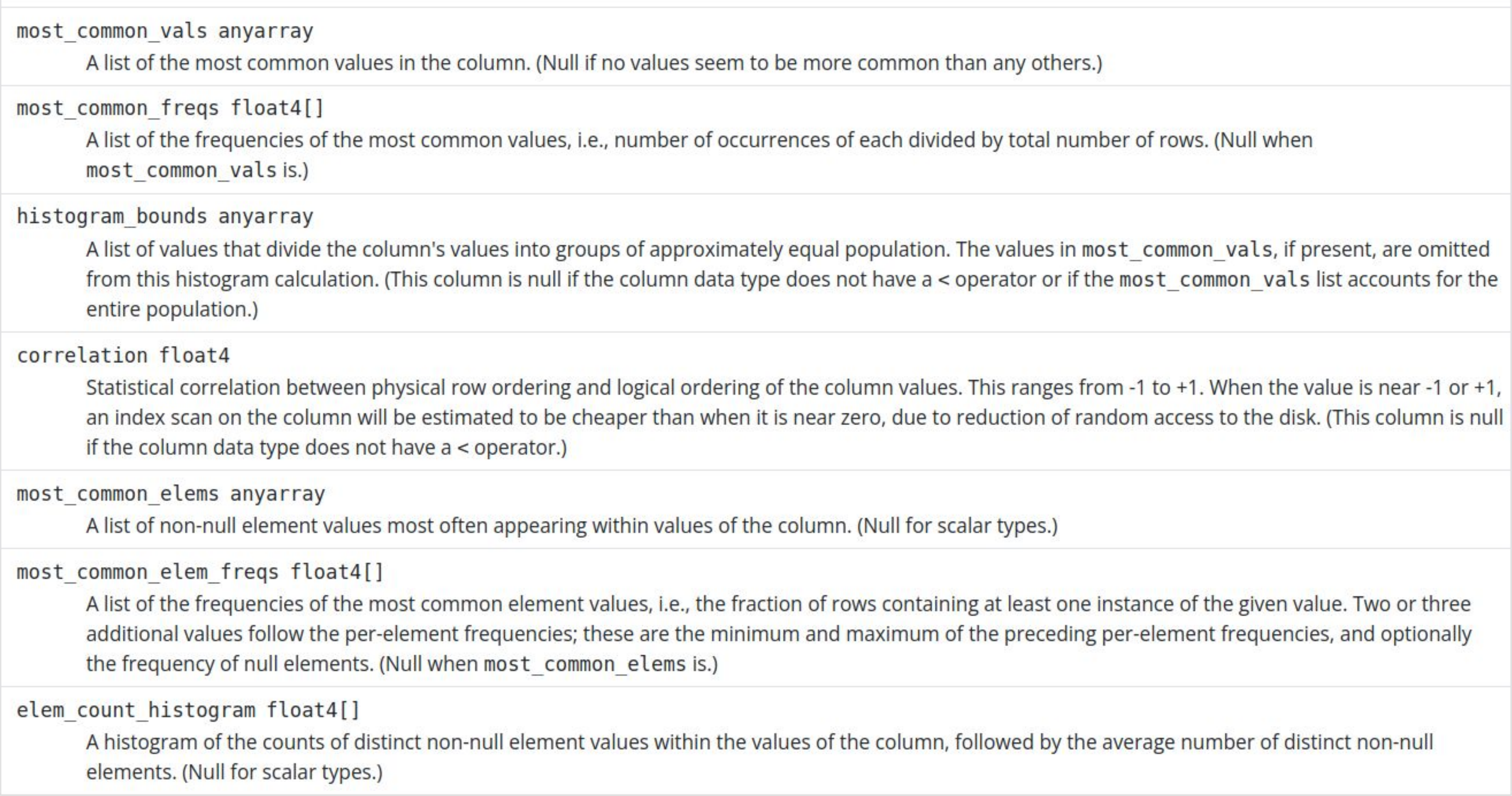


Table 51.89. pg_stats Columns

Column Type	Description
schemaname name (references pg_namespace.nspname)	Name of schema containing table
tablename name (references pg_class.relname)	Name of table
attname name (references pg_attribute.attname)	Name of the column described by this row
inherited bool	If true, this row includes inheritance child columns, not just the values in the specified table
null_frac float4	Fraction of column entries that are null
avg_width int4	Average width in bytes of column's entries
n_distinct float4	If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows. (The negated form is used when ANALYZE believes that the number of distinct values is likely to increase as the table grows; the positive form is used when the column seems to have a fixed number of possible values.) For example, -1 indicates a unique column in which the number of distinct values is the same as the number of rows.



`most_common_vals anyarray`

A list of the most common values in the column. (Null if no values seem to be more common than any others.)

`most_common_freqs float4[]`

A list of the frequencies of the most common values, i.e., number of occurrences of each divided by total number of rows. (Null when `most_common_vals` is.)

`histogram_bounds anyarray`

A list of values that divide the column's values into groups of approximately equal population. The values in `most_common_vals`, if present, are omitted from this histogram calculation. (This column is null if the column data type does not have a < operator or if the `most_common_vals` list accounts for the entire population.)

`correlation float4`

Statistical correlation between physical row ordering and logical ordering of the column values. This ranges from -1 to +1. When the value is near -1 or +1, an index scan on the column will be estimated to be cheaper than when it is near zero, due to reduction of random access to the disk. (This column is null if the column data type does not have a < operator.)

`most_common_elems anyarray`

A list of non-null element values most often appearing within values of the column. (Null for scalar types.)

`most_common_elem_freqs float4[]`

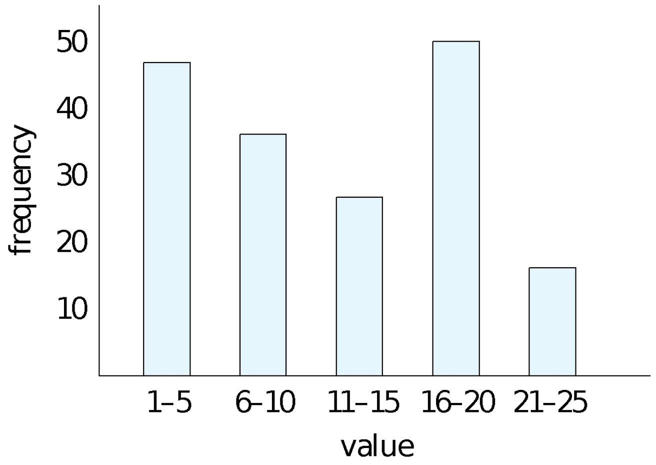
A list of the frequencies of the most common element values, i.e., the fraction of rows containing at least one instance of the given value. Two or three additional values follow the per-element frequencies; these are the minimum and maximum of the preceding per-element frequencies, and optionally the frequency of null elements. (Null when `most_common_elems` is.)

`elem_count_histogram float4[]`

A histogram of the counts of distinct non-null element values within the values of the column, followed by the average number of distinct non-null elements. (Null for scalar types.)

Histograms

- Histogram on attribute *age* of relation *person*



- **Equi-width** histograms
- **Equi-depth** histograms break up range such that each range has (approximately) the same number of tuples
  - E.g. (4, 8, 14, 19)
- Many databases also store *n* **most-frequent values** and their counts
  - Histogram is built on remaining values only

Histograms (cont.)

- Histograms and other statistics usually computed based on a **random sample**
- Statistics may be out of date
  - Some database require a **analyze (vacuum)** command to be executed to update statistics
  - Others automatically recompute statistics
    - e.g., when number of tuples in a relation changes by some percentage

Selection Size Estimation

$\sigma_{A=v}(r)$

- $n_r / V(A,r)$: number of records that will satisfy the selection
- Equality condition on a key attribute: *size estimate* = 1

$\sigma_{A \leq V}(r)$ (case of $\sigma_{A \geq V}(r)$ is symmetric)

- Let c denote the estimated number of tuples satisfying the condition.
- If min(A,r) and max(A,r) are available in catalog
  - c = 0 if v < min(A,r)
  - c = $n_r . \dfrac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information *c* is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

- The **selectivity** of a condition θ_i is the probability that a tuple in the relation *r* satisfies θ_i .
  - If s_i is the number of satisfying tuples in *r*, the selectivity of θ_i is given by s_i / n_r.
- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. *Assuming independence,* estimate of tuples in the result is: $n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$
- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Estimated number of tuples:

$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$

- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples: $n_r - size(\sigma_\theta(r))$

Join Operation: Running Example

Running example: *student* ⋈ *takes*

Catalog information for join examples:

- $n_{student}$ = 5,000.
- $f_{student}$ = *50,* which implies that
- $b_{student}$ =5000/50 = 100.
- n_{takes} = 10000.
- f_{takes} = 25, which implies that
- b_{takes} = 10000/25 = 400.
- *V(ID, takes)* = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
    - Attribute *ID* in *takes* is a foreign key referencing *student.*
    - *V*(*ID, student*) = 5000 (*primary key!*)

Estimation of the Size of Joins

- The Cartesian product *r* x *s* contains $n_r.n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.
- If $R \cap S = \emptyset$, then $r \bowtie s$ is the same as *r* x *s.*
- If $R \cap S$ is a key for *R*, then a tuple of *s* will join with at most one tuple from *r*
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in *s.*
- If $R \cap S$ *in* S is a foreign key in *S* referencing *R,* then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in *s.*
  - The case for $R \cap S$ being a foreign key referencing *S* is symmetric.
- In the example query *student* ⋈ *takes, ID* in *takes* is a foreign key referencing *student*
  - hence, the result has exactly n_{takes} tuples, which is 10000

Estimation of the Size of Joins (Cont.)

- If *R* ∩ *S* = {*A*} is not a key for *R* or *S*.
  If we assume that every tuple *t* in *R* produces tuples in *R*⋈*S*, the number of tuples in *R* ⋈ *S* is estimated to be:

  $$\frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

  $$\frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.
- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

Estimation of the Size of Joins (Cont.)

- Compute the size estimates for *depositor* ⋈ *customer* without using information about foreign keys:
  - *V(ID, takes)* = 2500, and
    V(ID, student) = 5000
  - The two estimates are 5000 * 10000/2500 = 20,000 and 5000 * 10000/5000 = 10000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\prod_A(r)$ = $V(A,r)$
- Aggregation : estimated size of ${}_G\gamma_A(r)$ = $V(G,r)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g., $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \text{ or } \theta 2}(r)$
  - For operations on different relations:
    - estimated size of $r \cup s$ = size of r + size of s.
    - estimated size of $r \cap s$ = minimum size of r and size of s.
    - estimated size of $r - s$ = r.
    - All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.

Size Estimation (Cont.)

- Outer join:
  - Estimated size of r ⟕ s = *size of* $r \bowtie s$ + *size of* r
    - Case of right outer join is symmetric
  - Estimated size of r ⟗ s = *size of* $r \bowtie s$ + size of r + size of s

Intermediate Results Number of Distinct Values

- Selections: $\sigma_{\theta}(r)$
- If θ forces A to take a specified value: $V(A,\sigma_{\theta}(r)) = 1$.
    - e.g., $A = 3$
- If θ forces A to take on one of a specified set of values:

  $V(A,\sigma_{\theta}(r))$ = number of specified values.
    - (e.g., ($A = 1 \vee A = 3 \vee A = 4$)),
- If the selection condition θ is of the form A *op* r

  estimated $V(A,\sigma_{\theta}(r)) = V(A.r) * s$
    - where s is the selectivity of the selection.
- In all the other cases: use approximate estimate of

  $\min(V(A,r), n_{\sigma\theta(r)})$
    - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

Joins: $r \bowtie s$

- If all attributes in A are from r

  estimated $V(A, r \bowtie s) = \min(V(A,r), n_{r \bowtie s})$

- If A contains attributes $A1$ from r and $A2$ from s, then estimated

  $V(A,r \bowtie s) =$

  - $\min(V(A1,r)*V(A2 - A1,s), V(A1 - A2,r)*V(A2,s), n_{r \bowtie s})$

  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

- Estimation of distinct values are straightforward for projections.
  - They are the same in $\prod_{A\ (r)}$ as in *r*.
- The same holds for grouping attributes of aggregation.
- For aggregated values
  - For min(*A*) and max(*A*), the number of distinct values can be estimated as min(V(*A*,*r*), *V*(*G*,*r*)) where G denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use *V*(*G*,*r*)

The Internals of PostgreSQL

Chapter 3

Query Processing

https://www.interdb.jp/pg/pgsql03.html

Postgres optimizer code snippets

Postgres genetic query optimizer

https://www.postgresql.org/docs/13/geqo-intro.html

https://doxygen.postgresql.org/geqo_8h_source.html

var=const selectivity

https://doxygen.postgresql.org/selfuncs_8h.html#a31ee9824c23028c56ca3d6ca92c39a7e

Range typanalyze

https://doxygen.postgresql.org/rangetypes__typanalyze_8c_source.html

Range overlap

https://github.com/postgres/postgres/blob/cd3f429d9565b2e5caf0980ea7c707e37bc3b317/src/include/catalog/pg_operator.dat#L3110

rangesel

https://doxygen.postgresql.org/rangetypes__selfuncs_8c.html#a632d39f45c72d18cf792fb33014155ee

Additional Optimization techniques

Materialized Views

- A **materialized view** is a view whose contents are computed and stored.
- Consider the view

```
create view department_total_salary(dept_name,
total_salary) as
    select dept_name, sum(salary)
    from instructor
    group by dept_name
```

- Materializing the above view would be very useful if the total salary by department is required frequently
    - Saves the effort of finding multiple tuples and adding up their amounts

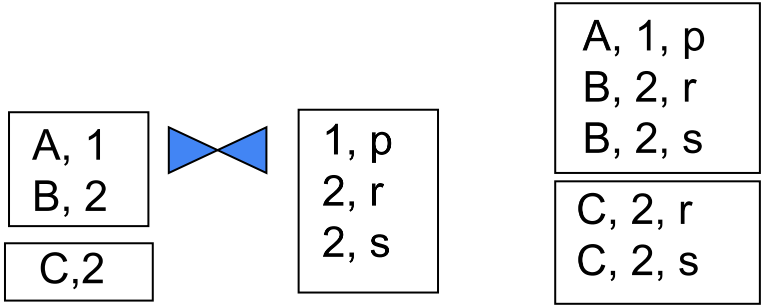
Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Recompute at every update ??? nooooo
- A better option is to use **incremental view maintenance**

- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Periodic recomputation (e.g. nightly)
  - Incremental maintenance supported by many database systems

Incremental View Maintenance - Join

- Consider the materialized view $v = r \bowtie s$ and an update to r
- Let r^{old} and r^{new} denote the old and new states of relation r
- Consider the case of an insert to r:
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to $(r^{\text{old}} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{\text{old}} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
- Thus, for inserts $v^{new} = v^{old} \cup (i_r \bowtie s)$
- Similarly for deletes $v^{new} = v^{old} - (d_r \bowtie s)$



A, 1
B, 2

C,2

1, p
2, r
2, s

A, 1, p
B, 2, r
B, 2, s

C, 2, r
C, 2, s

View Maintenance - Selection and Projection Operations

- Selection: Consider a view $v = \sigma_\theta(r)$.
  - $v^{new} = v^{old} \cup \sigma_\theta(i_r)$
  - $v^{new} = v^{old} - \sigma_\theta(d_r)$
- Projection is a more tricky
  - r(A, B) = { (a,2), (a,3)}, $v = \prod_A(r)$
  - If we delete the tuple (a,2) from r:
    - DELETE FROM v WHERE A=a --wrong
- For each tuple in a projection $\prod_A(r)$, we will keep a count of how many times it was derived
  - On insert of a tuple to r, if the resultant tuple is already in $\prod_A(r)$ we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from r, we decrement the count of the corresponding tuple in $\prod_A(r)$
    - if the count becomes 0, we delete the tuple from $\prod_A(r)$

View Maintenance - Aggregation Operations

Count : $v = {}_{A}\gamma_{count(B)}(r)$.

- When a set of tuples i_r is inserted
  - For each tuple r in i_r, if the corresponding group is already present in v, we increment its count, else we add a new tuple with count = 1
- When a set of tuples d_r is deleted
  - for each tuple t in i_r. we look for the group *t.A* in *v*, and subtract 1 from the count for the group.
    - If the count becomes 0, we delete from *v* the tuple for the group *t.A*

Sum: $v = {}_{A}\gamma_{sum\ (B)}(r)$

- Similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
- Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from v

Incremental View Maintenance - Aggregate Operations

- **Avg**: How to handle average?
  - Maintain **sum** and **count** separately, and divide at the end
- **min**, **max**: $v = {}_{A}\gamma_{min\,(B)}(r)$.
  - Handling insertions on r is straightforward.
  - Maintaining the aggregate values **min** and **max** on deletions may be more expensive. We have to look at the other tuples of *r* that are in the same group to find the new minimum

View Maintenance - Other Operations

- Set intersection: $v = r \cap s$
  - When a tuple is inserted in *r* we check if it is present in *s*, and if so we add it to *v*.
  - If the tuple is deleted from r, we delete it from the intersection if it is present.
  - Updates to *s* are symmetric
  - The other set operations, *union* and *set difference* are handled in a similar fashion.

Query Optimization and Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view $v = r \bowtie s$ is available
  - A user submits a query $r \bowtie s \bowtie t$
  - We can rewrite the query as $v \bowtie t$
    - Whether to do so depends on cost estimates for the two alternative
- Replacing a use of a materialized view by the view definition:
  - A materialized view $\mathrm{v} = \mathrm{r} \bowtie \mathrm{s}$ is available, but without any index on it
  - User submits a query $\sigma_{A=10}(\mathrm{v})$.
  - Suppose also that *s* has an index on the common attribute B, and r has an index on attribute A.
  - The best plan for this query may be to replace *v* by $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(\mathrm{r}) \bowtie \mathrm{s}$
- Query optimizer should be extended to consider all above alternatives and choose the best overall plan

Materialized View Selection

- **Materialized view selection**: “What is the best set of views to materialize?”
- **Index selection**: “what is the best set of indices to create”
  - closely related, to materialized view selection but simpler
- Materialized view selection and index selection based on typical system **workload** (queries and updates)
  - Typical goal: minimize time to execute workload, subject to constraints on space and time taken for some critical queries/updates
- Commercial database systems provide tools (called “tuning assistants” or “wizards”) to help the database administrator choose what indices and materialized views to create

Optimization of Updates

Halloween problem

```
update R set A = 5 * A
where A > 10
```

- If index on A is used to find tuples satisfying A > 10, and tuples updated immediately, same tuple may be found (and updated) multiple times
- Solution 1: *Always defer updates*
  - collect the updates (old and new values of tuples) and update relation and indices in second pass
  - Drawback: extra overhead even if e.g. update is only on R.B, not on attributes in selection condition
- Solution 2: *Defer only if required*

Multiquery Optimization

Example

```
Q1: select * from (r natural join t) natural join s
Q2: select * from (r natural join u) natural join s
```

- Both queries share common subexpression (r natural join s)
- May be useful to compute (r natural join s) once and use it in both queries

- **Multiquery optimization**: find best overall plan for a set of queries

Parametric Query Optimization

Example

```
select *
from r natural join s
where r.a < $1
```

- value of parameter $1 not known at compile time
    - known only at run time
- different plans may be optimal for different values of $1
- Solution 1: optimize at run time, each time query is submitted
        - can be expensive
- Solution 2: **Query Plan Caching & Parametric Query Optimization**:
    - optimizer generates a set of plans, optimal for different values of $1
    - Choose at runtime, when $1 is known

Credits

Many slides in this lecture are taken from:

- Avi Silberschatz, Henry F. Korth, S. Sudarshan. Database System Concepts

Recommended reading

- The Internals of PostgreSQL (https://www.interdb.jp/pg/)