PostgreSQL Statistics Injection Final Presentation

Ethan Lin, Zhiping Liao

Goals & Schedule

75% goal Done: Develop API functions to **export** PostgreSQL's statistics and system catalog to a portable and version-independent format (maybe JSON), and later **import** statistics to a PostgreSQL database.

100% goal Done: Use the exported statistics and data samples to **estimate joint distributions** of certain columns, and inject the estimated distribution back to the database for better query optimization performance.

125% goal: WAL listener to modify gradually modify statistics for more up-to-date statistics

Background: Postgres Histograms

Histograms are created by sampling tuples (by default 30,000) from tables, sort values in one column and split the values into equi-width buckets (by default 100).

- Most common values are first excluded from samples, making histograms a tail-distribution only
- Hard to use it to derive other statistics directly

PostgreSQL only supports single-column and multi-column statistics within one table

VARCHAR columns are simply sorted in lexical order; no sense of “being continuous”

Method: Test Column Correlation within Table

1. Similar to Postgres, sample a batch of tuples from each table.

2. Sort the values from each column in order and construct a histogram for each column

3. Detect correlation by analyzing histograms from any column pair of the table

Method: Test Correlation with KL Divergence

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)}$$

P, Q: Distributions of two random variables

x: Any value these two distributions can take

Measures: “Difference” between two distributions

Expected distribution when two columns are independent:

- Cartesian product of the ranges of two column distributions, uniform distribution

Actual distribution observed from samples:

- $P(x_1,\ x_2) = \frac{(x_1, x_2)\ in\ samples}{\#samples}$

Method: Classical Statistical Tools

Chi-Square Test: Test whether two categorical variables are independent

Spearman Test: Assesses how well the relationship between two variables is monotonic

Pearson Test: Measures linear relationship between two variables

Experiment Results

We ran experiments on TPC-DS benchmark with scale factor 1. We gather KL divergence values from all column pairs and sort them in decreasing order.

Capable of detecting facts like:

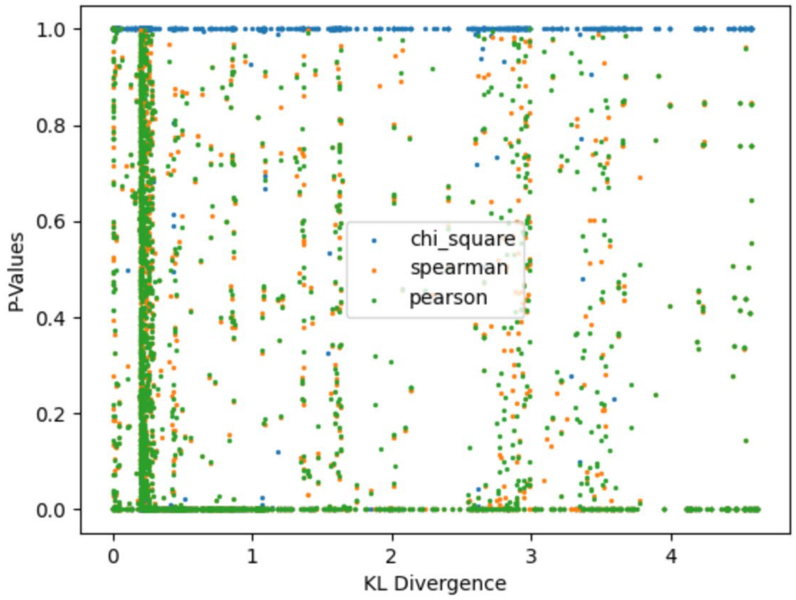
- Year = Financial Year
- Refunding Address = Returning Address
- Paid amount is related to paid taxes

	col1	col2	kl
0	d_year	d_fy_year	4.619392477646311
1	d_quarter_seq	d_fy_quarter_seq	4.60665347976636
2	d_month_seq	d_first_dom	4.605236820901854
3	d_week_seq	d_fy_week_seq	4.6050721009512925
4	d_date_sk	d_date	4.60505676854301
5	d_date_sk	d_same_day_ly	4.60505676854301
6	d_date_sk	d_same_day_lq	4.60505676854301
7	d_date	d_same_day_ly	4.60505676854301
8	d_date	d_same_day_lq	4.60505676854301
9	d_same_day_ly	d_same_day_lq	4.60505676854301
10	t_time_sk	t_time	4.60505676854301
11	d_date_sk	d_week_seq	4.579286902816279
12	d_date_sk	d_fy_week_seq	4.579286902816279
13	d_date	d_week_seq	4.579286902816279
14	d_date	d_fy_week_seq	4.579286902816279
15	d_week_seq	d_same_day_ly	4.579286902816279
16	d_week_seq	d_same_day_lq	4.579286902816279
17	d_fy_week_seq	d_same_day_ly	4.579286902816279
18	d_fy_week_seq	d_same_day_lq	4.579286902816279
19	d_month_seq	d_week_seq	4.522463139436349
20	d_month_seq	d_fy_week_seq	4.522463139436349
21	d_week_seq	d_first_dom	4.522463139436349
22	d_fy_week_seq	d_first_dom	4.522463139436349
23	d_date_sk	d_month_seq	4.50929149394438

Experiment Results

While KL divergence gives meaningful results, classical statistical tools are confused

- P-value either 0 or 1
- Basically not related to KL divergence
- Explanation?



Further Experiment Setting?

Understand the effects of building statistics on “most correlated” columns vs. “least correlated” columns on query optimizers

1. For one table, sort column pairs by KL divergence values
2. Split column pairs into N consecutive groups
3. For each group, build multivariate statistics on all column pairs, and run EXPLAIN on TPC-DS queries.
4. Compare the results with EXPLAIN ANALYZE results.

Problem: Each group gives identical EXPLAIN results.

`pg_statistic_ext` extraction and injections

Pretty tedious: read C code, write in Rust, basically

Caveat:

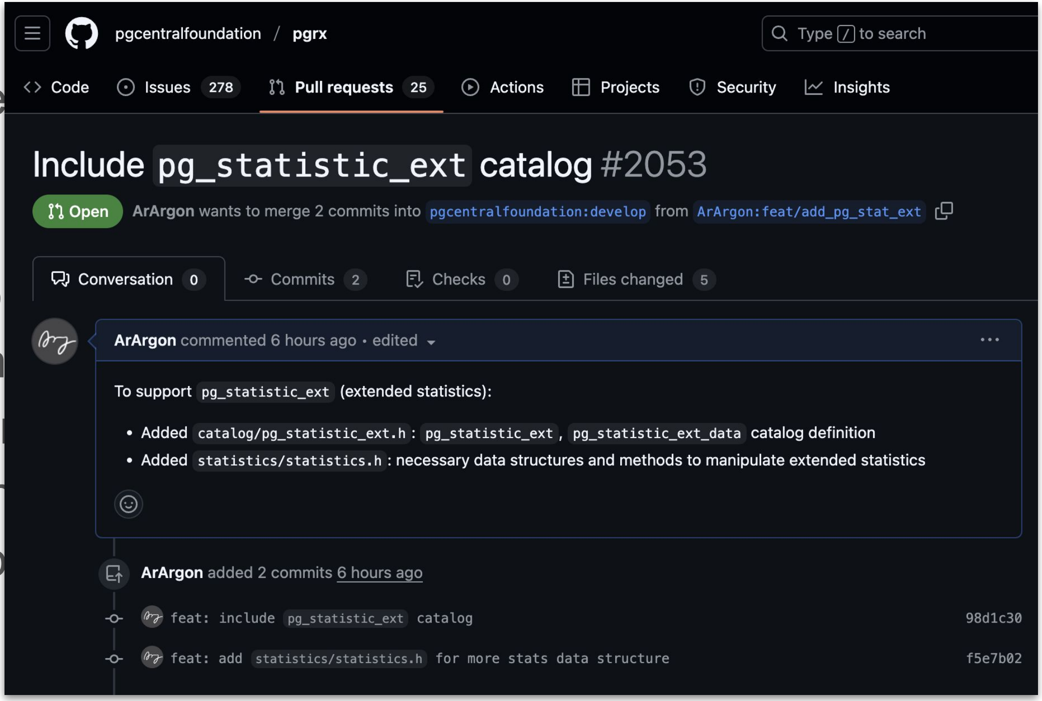
- Datum Import / Export;
- Variable Length Fields
- MCV-family records are **serialized**, not flattened fields
- `Pgrx` does not bind extended statistics 👉 PR
- Stats on expressions are not supported:
  `pg_node_tree` 👉 internal expression tree

`pg_statistic_ext` extraction and injections

Pretty tedious: re

Caveat:

- Datum Impo
- Variable Len
- MCV-family
- `Pgrx` does r
- Stats on exp `pg_node_t`



...nd injections

```c
extern MVNDistinct *statext_ndistinct_load(Oid mvoid, bool inh);
extern MVDependencies *statext_dependencies_load(Oid mvoid, bool inh);
extern MCVList *statext_mcv_load(Oid mvoid, bool inh);

extern void BuildRelationExtStatistics(Relation onerel, bool inh, double totalrows,
                                       int numrows, HeapTuple *rows,
                                       int natts, VacAttrStats **vacattrstats);
extern int  ComputeExtStatisticsRows(Relation onerel,
                                     int natts, VacAttrStats **vacattrstats);
```

F... ...ally

C...

- Datum Import / Export:
-
- ...not fla...
- ...ics 👉
- ...ted:
- ...sion tre...

```c
CATALOG(pg_statistic_ext_data,3429,StatisticExtDataRelationId)
{
    Oid         stxoid BKI_LOOKUP(pg_statistic_ext);    /* statistics object
                                                         * this data is for */
    bool        stxdinherit;    /* true if inheritance children are included */

#ifdef CATALOG_VARLEN           /* variable-length fields start here */

    pg_ndistinct stxdndistinct; /* ndistinct coefficients (serialized) */
    pg_dependencies stxddependencies;   /* dependencies (serialized) */
    pg_mcv_list stxdmcv;        /* MCV (serialized) */
    pg_statistic stxdexpr[1];   /* stats for expressions */

#endif  #ifdef CATALOG_VARLEN

} FormData_pg_statistic_ext_data;
```

```c
/*
 * Multivariate MCV (most-common value) lists
 *
 * A straightforward extension of MCV items - i.e. a list (array) of
 * combinations of attribute values, together with a frequency and null flags.
 */
typedef struct MCVItem
{
    double      frequency;      /* frequency of this combination */
    double      base_frequency; /* frequency if independent */
    bool       *isnull;         /* NULL flags */
    Datum      *values;         /* item values */
} MCVItem;

/* multivariate MCV list - essentially an array of MCV items */
typedef struct MCVList
{
    uint32      magic;          /* magic constant marker */
    uint32      type;           /* type of MCV list (BASIC) */
    uint32      nitems;         /* number of MCV items in the array */
    AttrNumber  ndimensions;    /* number of dimensions */
    Oid         types[STATS_MAX_DIMENSIONS];    /* OIDs of data types */
    MCVItem     items[FLEXIBLE_ARRAY_MEMBER];   /* array of MCV items */
} MCVList;
```

How good is `pg_statistic_ext`?

- Well, …
- Our initial experiment compared query plan differences before and after adding statistics by group on 2 tables. Almost all of them were the same.
- Experiment: add all column pairs on the following tables: `call_center`, `catalog_page`, `catalog_returns`, `catalog_sales`, `customer`, `customer_address`, `customer_demographics`,
- Total 152 queries, 1179 statistics
- 15 queries entered `statext_clauselist_selectivity`, 13 tried MCV, 11 actually applied MCV on 3 statistics

How does PostgreSQL utilize extended statistics?

Use	Build
`make_join_rel`	`vacuum`
→ `build_join_rel`	→ `analyze_rel`
→ `set_joinrel_size_estimates`	→ `do_analyze_rel`
→ `calc_joinrel_size_estimate`	→ `BuildRelationExtStatistics`
→ `clauselist_selectivity`	→ **`statext_mcv_build`**
→ **`statext_clauselist_selectivity`**	

How does PostgreSQL utilize extended statistics?

What could go wrong?

Use

make_join_rel

→ build_join_rel

→ set_joinrel_si

→ calc_joinrel_s

→ clauselist_sel

→ **statext_clauselist_selectivity**

```
/*
 * Determine if these clauses reference a single relation.  If so, and if
 * it has extended statistics, try to apply those.
 */
rel = find_single_rel_for_clauses(root, clauses);
if (use_extended_stats && rel && rel->rtekind == RTE_RELATION && rel->statlist != NIL)
{
    /*
     * Estimate as many clauses as possible using extended statistics.
     *
     * 'estimatedclauses' is populated with the 0-based list position
     * index of clauses estimated here, and that should be ignored below.
     */
    s1 = statext_clauselist_selectivity(root, clauses, varRelid,
                                        jointype, sjinfo, rel,
                                        &estimatedclauses,  is_or: false);
}
```

Build

vacuum

el

e_rel

tionExtStatistics

cv_build

How does PostgreSQL utilize extended statistics?

1. First try most common values (MCV)
   a. Greedily try every available MCV
2. Then, try functional dependencies

```c
/* sum frequencies for all the matching MCV items */
*basesel = 0.0;
*totalsel = 0.0;
for (i = 0; i < mcv->nitems; i++)
{
    *totalsel += mcv->items[i].frequency;

    if (matches[i] != false)
    {
        *basesel += mcv->items[i].base_frequency;
        s += mcv->items[i].frequency;
    }
}

return s;
```

```c
Selectivity sel;

/* First, try estimating clauses using a multivariate MCV list. */
sel = statext_mcv_clauselist_selectivity(root, clauses, varRelid, jointype,
                                         sjinfo, rel, estimatedclauses, is_or);

/*
 * Functional dependencies only work for clauses connected by AND, so for
 * OR clauses we're done.
 */
if (is_or)
    return sel;

/*
 * Then, apply functional dependencies on the remaining clauses by calling
 * dependencies_clauselist_selectivity.  Pass 'estimatedclauses' so the
 * function can properly skip clauses already estimated above.
 *
 * The reasoning for applying dependencies last is that the more complex
 * stats can track more complex correlations between the attributes, and
 * so may be considered more reliable.
 *
 * For example, MCV list can give us an exact selectivity for values in
 * two columns, while functional dependencies can only provide information
 * about the overall strength of the dependency.
 */
sel *= dependencies_clauselist_selectivity(root, clauses, varRelid,
                                           jointype, sjinfo, rel,
                                           estimatedclauses);

return sel;
```

Takeaways

- Pg’s extended statistics are quite limited in improving cardinality estimation.
    - Sketches, adaptive stats require major overhaul to Postgres’ planner, cost model
    - `add_paths_to_joinrel`: called after a join path is costed (built).
    - `set_rel_pathlist_hook`: called after a scan filters.
- Query-oriented / adaptive approach might yield better results
- Correlation between join predicates is far more important than correlation within one table, especially when norm form is high
- Database joint distributions are sparse; classical statistical tools work badly under this assumption