Extending PDQ to extract statistics from PostgreSQL

Dr. Anna Roubíčková, Dr. Michael Jackson
{a.roubickova, m.jackson}@epcc.ed.ac.uk
EPCC, University of Edinburgh

Introduction

PDQ (Proof-Driven Querying)[1] is a Java library implementing a proof-driven approach to generating and optimizing plans for answering queries, especially over various datasources that are semantically linked, but may have different access interfaces. PDQ can take into account such things as access restrictions or different costs. The optimisation is done by converting the query to a logically equivalent first-order logic formula and generating a (set of) proof(s) in the model defined by the database schema, its constraints and the access interfaces. Every such proof corresponds to a plan for answering the query and has associated costs which allow to choose the optimal one.

Query optimisation relies heavily on estimating cardinality of intermediate products the system builds in order to construct the answer, such as the amount of tuples returned by `SELECT` or `JOIN` operations. This is typically estimated based on statistical information about relations involved in the query and its answer. Better cardinality estimation allows the planner to find more optimal ways to answer queries.

PDQ can load statistical information about relations from the SQL Server[2]. Our work focused on linking PDQ with PostgreSQL[3] in a similar manner. The code was developed in a separate branch of the PDQ repository[4].

This work was done in collaboration with Dr. Efthymia Tsamoura and Benjamin Spencer of the University of Oxford and was funded by Scottish Enterprise as part of the Alan Turing Institute-Scottish Enterprise Data Engineering Program.

[1]http://www.cs.ox.ac.uk/projects/pdq/home.html
[2]https://www.microsoft.com/en-gb/sql-server
[3]https://www.postgresql.org
[4]https://github.com/alan-turing-institute/pdq/tree/feature-postgres-histograms

1 PDQ

PDQ has some histogram functionality implemented for an SQL Server backend, our aim is to provide a similar functionality for Postgresql. To that end, have a look at the default Postgres behaviour and the statistical information it collects (Appx. A), we'll present the attempt to understand the SQL Server functionality (Appx. B), and compare the two. We touch on the theory underlying the sampling and histogram construction in Sec. 3.

1.1 SQLServer

Currently, PDQ has methods to load a histogram from SQL Server. The code can be found in the `cost` package, in its `statistics` part:
`cost/main/src/uk/ac/ox/cs/pdq/cost/statistics`

The `SQLServerHistogramLoader` class provides the infrastructure to create and populate a `SQLServerHistogram` object. The loader function `load`, given a name of a file, loads the histogram into the object. I have found histogram files (`V1Histogram.rpt, V2Histogram.rpt`) in `cost/test/src/uk/ac/ox/cs/pdq/test/cost/estimators/statistics/estimators/input/` but I have not investigated where these came from.

The histogram file is read line-by-line, and from each line it retrieves information about one bucket, converts the strings (read from file) to the appropriate Java type based on the attribute's type and creates the bucket object that is then added to the list of this histogram's buckets.

The SQLServerBucket class. The data for each attribute (a column of a table) are gathered in buckets:

```
public class SQLServerBucket implements Bucket{
   /** data **/
   private final Object range_hi_key;
   private final BigInteger range_rows;
   private final BigInteger eq_rows;
   private final BigInteger distinct_range_rows;
   private final double avg_range_rows;

   /** bucket operations **/
   public SQLServerBucket(Object range_hi_key, BigInteger
      range_rows, BigInteger eq_rows, BigInteger
      distinct_range_rows, double avg_range_rows)
   public Object getRange_hi_key()
```

```
    public BigInteger getRange_rows()
    public BigInteger getEq_rows()
    public BigInteger getDistinct_range_rows()
    public double getAvg_range_rows()
    public BigInteger getNumberOfElements()
    public double getVariance()
    public String toString()
    public boolean equals(Object o)
    public int hashCode()
}
```

Every bucket is specified by its bound(s) — here, the upper bound is represented by `range_hi_key` and it is included in the bucket. The lower bound of a bucket is the upper bound of the previous bucket and is excluded from the current bucket. E.g., a histogram with 10 buckets would be represented by bounds $b_1, \ldots, b_{10}$ with the buckets $(-\infty, b_1], (b_1, b_2], \ldots, (b_9, b_{10}]$. [5]

The bucket records how many rows have the same value as the upper bound (`eq_rows`) and how many other values fall into the bucket (`range_rows`), how many rows have unique values (`distinct_range_rows`) and, on average, how frequently duplicate values appear (`avg_range_rows`). Note that the `avg_range_rows` calculation excludes values equal to upper bound.

The bucket methods construct the object, retrieve the above information, and also:

- retrieve the number of elements in the bucket (as number of rows in the bucket plus the number of rows whose value equals to the upper bound)
- retrieve the bucket's variance[6]
- convert the bucket to a string (for printing)
- test the bucket for equivalence to another bucket
- get a hash code for a bucket

The SQLServerHistogram class. The histogram object holds all the `buckets` and statistical information about the histogram (`samplingFactor`,

[5]In some implementations, only $n-1$ separating values $b_1, \ldots, b_{n-1}$ are given to represent n buckets, and $b_n = +\infty$ is assumed.

[6]The bucket's variance is (a) not implemented (it currently returns 0), and (b) needed for identifying bucket with the largest variance (in `SQLServerHistogram`), which at this point doesn't really do anything.

bucketWithLargestVariance). The histogram methods make the information accessible and further operations with buckets are available.

```
public class SQLServerHistogram implements Histogram{
   /** data **/
   private final List<SQLServerBucket> buckets;
   private final SQLServerBucket bucketWithLargestVariance;
   private final double samplingFactor = 0.15;

   /** histogram methods **/
   public SQLServerHistogram(List<SQLServerBucket> buckets)
   public Bucket getBucketWithLargestVariance()
   public double getSamplingFactor()
   public String toString()

   /** operations with buckets **/
   public List<SQLServerBucket> getBuckets()
   public SQLServerBucket getBucket(int bucketIndex)
   public SQLServerBucket getBucket(Object value)
   public int hashCode()
   public boolean equals(Object o)

   /** cardinality estimation **/
   public BigInteger
      estimateSingleJoinAttributeCardinality(Histogram input)
}
```

It is not entirely clear where the samplingFactor of 0.15 comes from. Also, bucketWithLargestVariance is lacking an implementation. The former is probably dependent on the SQL Server implementation and might be possible to retrieve by querying SQL Server; the latter needs a sound definition of variance for attributes of different types.

The intriguing function is `estimateSingleJoinAttributeCardinality`. It belongs to the `SQLServerJoinCardinalityEstimator` class, which contains more cardinality estimators and support functions. This one estimates single join attribute cardinality and is described in the code as follows:

```
public BigInteger estimateSingleJoinAttributeCardinality
    (SQLServerHistogram left, SQLServerHistogram right)
```

`@param`	`left`	the left histogram
`@param`	`right`	the right histogram
`@return`		the estimated cardinality of the `left` and `right` join

Let the buckets of the histogram are the same after the bucket alignment, namely $B = \{b_1, ..., b_K\}$. The returned estimate is

$$\begin{aligned} \text{SizeOf}(\text{AnnPlan}) = \sum_{i \le K} & \text{AvgSize}(R_1[b_i]) \times \text{AvgSize}(R_2[b_i]) \times \\ & \times \min\{\text{NumDistinct}(R_1[b_i]), \text{NumDistinct}(R_2[b_i])\}, \end{aligned} \tag{1}$$

where AvgSize($R_j[b_i]$) is the average number of tuples in R_j per element in the bucket b_i, and
NumDistinct($R_j[b_i]$) is the number of distinct values of the position corresponding to chase constant c in bucket b_i of R_j.

Very course histogram alignment takes place; a bucket b from the first histogram is aligned with a set of buckets B from the second histogram if b intersects with B. The cardinality estimation algorithm relies on the containment assumption.

I assume that R_1 and R_2 are the attributes that `left` and `right` histograms represent.
In the code, $\text{AvgSize}(R_j[b_i]) = \frac{\text{total number of rows in the bucket}}{\text{number of distinct rows in the bucket}}$.

Issues

- **Sampling factor.** In the current histogram object the sampling factor is hard-coded to 0.15. From the sampling theory [?] we know that there is an trade-off between sampling factor, size of the data and desired error. It is not clear why 0.15 is chosen, and depending on the data size it may result in less precise histograms. Further information is in Appx. B.

- **Variance.** It is not clear what the variance is, it is not implemented for SQL Server, and the method is not called anywhere (apart from deciding `bucketWithLargestVariance`).

1.2 Postgres

Our aim was to implement a similar functionality for a Postgres back-end. Following the `SQLServerHistogram` code, we created `PostgresHistoram` and

PostgresBucket classes. In addition, due to the way Postgres stores and makes its statistics accessible, we also created a PostgresTableHistogram object.

The communication between Postgres and Java goes via the JDBC[7] API. The java functions construct a string representing the desired query and with a connection open to the database can execute such query statement. The answer to the query is returned in a ResultSet. The ResultSet is an interface that allows addressing parts of the answer by their field (or attribute) names (as stated in the SELECT list). The contents of the ResultSet need to be fetched using its access methods get*Type*, and the types need to match the types of the values in the database. The tricky ones are arrays, which are returned as anyarray and can be accessed only as a string, even though on abstract level we know they are a sequence of values. The work-around is to ask Postgres to cast the array to text and then to text array, which allows it to be accessed in java as an array of strings (see getPgStatsQueryStatement() in PostgresHistogramTableLoader).

The PostgresTableHistogram class. The Postgres pg_stats database table (see Appx. A) contains statistical information about the specified relation/table and querying it returns a ResultSet. Reading through it row by row, we build a histogram object for every attribute. The histograms are gathered in a PostgresTableHistogram object.

```java
public class PostgresTableHistogram {
   /** data **/
   private final String relation;
   private final Map<String, PostgresHistogram> histograms;

   /** methods **/
   public PostgresTableHistogram(String relation)
   public String getRelation()
   public Set<String> getAttributes()
   public int numAttributes()
   public PostgresHistogram getHistogram(String attribute)
   public void addHistogram(String attribute, PostgresHistogram
       histogram)

   public void print()
}
```

[7]https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

The loader function `PostgresTableHistogramLoader` loads the whole relation and its histograms at once. The `print()` functions were implemented with this in mind and can print the overview of the whole table at once.

The PostgresHistogram class. The histogram object gathers information about the attribute as a whole. For the time being it is filled with the information readily available in Postgres and doesn't match fully the PDQ's SQL Server behaviour. The official description of the statistics kept by Postgres is detailed in Tables 1, 2 in Appx. A.

```
public class PostgresHistogram implements Histogram {
  /** data **/
  private final String relation;
  private final String attribute;
  private int sqlType;
  private final List<PostgresBucket> buckets;
  private String[] histogramBounds;
  private double nullFraction;
  private double numDistinctValues;
  private String[] commonValues;
  private Float[] commonFreqs;
  private Float correlation = null;
  private PostgresBucket bucketWithLargestVariance = null;
  private double samplingFactor = -1;

  /** setting up the object/values **/
  public PostgresHistogram(String relation, String attribute, int
      sqlType)
  public void addBucket(PostgresBucket bucket)
  public int setSqlType()
  public void setBucketWithLargestVariance()
  public void setHistogramBounds(String[] histogramBounds)
  public void setNullFraction(double nullFraction)
  public void setNumDistinctValues(double numDistinctValues)
  public void setCommonValuesFreqs(String[] commonValues, Float[]
      commonFreqs)
  public void setCorrelation(Float correlation)
  public void setSamplingFactor(double samplingFactor)

  /** access methods **/
  public String getRelation()
  public String getAttribute()
  public String[] getHistogramBounds()
```

```
   public double getNullFraction()
   public double getNumDistinctValues()
   public String[] getCommonValues()
   public Float[] getCommonFreqs()
   public Float getCorrelation()
   public int getNumberOfBuckets()
   public PostgresBucket getBucket(int index)
   public boolean bucketsContainFrequentValues()

   /** inherited functionality **/
   public Bucket getLargestBucket()
   public Bucket getBucketWithLargestVariance()
   public double getSamplingFactor()
   public BigInteger
      estimateSingleJoinAttributeCardinality(Histogram input)

   public void print()
}
```

The object still contains a list of buckets, and has place-holders for `samplingFactor` and `bucketWithLargestVariance` once the issues surrounding them in the SQL Server implementation are resolved.

In addition, `PostgresHistogram` knows the relation- and attribute-name it corresponds to, types of values it holds, correlation and information about distribution of the values: `nullFraction`, `numDistinctValues`, `histogramBounds`, `commonValues` and `commonFreqs`.

There are methods to set all these variables as well as to retrieve the values stored within. In addition, there are methods to retrieve information about the buckets (their number, getting i-th bucket, getting special buckets (largest one, the one with largest variance)). Most of the inherited functionality is not implemented yet as its intended meaning isn't clear.

The PostgresBucket class. PostgresBucket class implements the buckets of the histogram. The buckets gather basic information about the attribute's values, such as the separator values, cardinalities etc.

```
public class PostgresBucket implements Bucket {
   /** data **/
   private final Object lowerBound;
   private final Object upperBound;
   private final int numRows;
   private final int numDistinctRows;
   private final double averageRows;
```

```
    public PostgresBucket(Object lowerBound, Object upperBound, int
        numRows, int numDistinctRows)

    /** access methods **/
    public Object getLowerBound()
    public Object getUpperBound()
    public int getNumRows()
    public int getNumDistinctRows()
    public double getAverageRows()
    public BigInteger getNumberOfElements()
    public double getVariance()

    public void print()
}
```

The Postgres bucket has an explicitly-declared lower bound; note however that the lowest value elements are *included* in the first bucket, but *excluded* in all the other buckets. The elements' count reflects this behaviour (and so there are two different queries for the row-count).

The methods allow access to the data stored within a bucket; in addition, `getNumberOfElements` implements the `Bucket` interface functionality and its behaviour coincides with `getNumRows`. `getVariance` returns 0 (in line with SQLServer implementation). `print()` replaces the `toString` function and prints out the bucket's information in human readable form.

Issues

- **numRows.** Postgres doesn't provide information about absolute counts, only relative amounts (eg. percentual frequencies of occurrences). Knowing the number of buckets, the number of rows in a table, nulls and frequent values, one can probably work out an average size of a bucket. Currently, we query the database to count the *exact* number of rows that fall in each bucket so that the `numRows` can be set properly. It is quite slow, but very precise!

  The queries are constructed by `getInclusiveBucketQueryStatement` and `getExclusiveBucketQueryStatement` helper functions which build the query.

- **Common elements.** There are entries in the `pg_stats` table related to *elements*: `most_common_elems`, `most_common_elem_freqs` and `elem_count_histogram`. For the instances we have, these come back as `null` and we don't know their intended meaning/function, though we suspect that, if present, they may be of use. We suspect these may come in play with attribute types that do not support ordering (such `BLOB`s etc.).

2 Points of note.

2.1 Number of Buckets.

It seems that SQL Server decides the optimal number of buckets (that is, histogram steps) on the fly (see Appx. B), whereas Postgres uses a default number of buckets (this can be set by the user, for a specific table (and a specific column):
`ALTER TABLE tablename ALTER COLUMN columnname SET STATISTICS` *target*, where target can be set in the range 0 to 10000, and -1 will revert to using the system default statistics target). We have seen 100 and 1000 as defaults in different versions of Postgres; it seems that SQL Server's default/ maximum is 200.

2.2 Frequent Values.

Postgres is doing something funny with frequent values: The aim is to construct an equi-height histogram (that is, the buckets hold the same or similar number of elements). If there are values that seem to be appearing more frequently than others, they may misshape the histogram, or make its construction impossible. Therefore, in Postgres, the frequent values are *excluded* from the histogram and stored separately as `String[] commonValues`. It is also useful to know the frequency with which these values appear, this is stored in `Float[] commonFreq`.

In some cases, there are only frequent values, those get excluded and the histogram cannot be constructed at all. (This can happen for example when the attribute holds information about category of goods, or country — these would typically have only few distinct values that would repeat over and over).

For attributes with only frequent values we reconstruct the histogram with as many buckets as there are the frequent values. Obviously, there are no guarantees about equi-heightness of such a histogram.

For the "mixed" attributes that have both a histogram and a list of frequent values, we keep the buckets of the Postgres histogram, but adjust the counts of elements to contain also the frequent values. This may obviously misshape the histogram again.

A good indicator is `bucketsContainFrequentValues` variable, which is `True` when frequent values are present in the histogram and hence the histogram is not guaranteed to have any sensible shape.

3 Theory

When constructing a histogram, the DBMS may choose to sample the data to speed up the process. There are two related question: (1) how do we choose the sample and (2) how big a sample do we need? There are two publications that offer answers to these: The theory of histogram construction (ad 1) comes from Jeff Vitter [?], the reasonable size of the sample (ad 2) is studied by Chaudhuri et al.[?].

3.1 Sampling Algorithm

Jeff [?] describes how to collect a sample from the data in one pass, even without knowing how large the data is. I am not sure how useful the "without knowing how large the data is" property is; however, awareness of this work is crucial as Postgres claims to be using it: In `postgresql-10.5/ src/backend/commands/analyze.c`[8] the function `acquire_sample_rows`, line `1155` — `if (sample_it)` calls back onto this paper [?]. The preceding code seems to be concerned with sampling and selectively (not) using the rest of the data on the loaded page.

This algorithm reads the data sequentially and at most once. According to Jeff's Thm. 1 [?] any such algorithm is a variation on a *Reservoir Algorithm*, i.e., the algorithm maintains a reservoir containing a random sample of the data processed so far from which the representative sample of the desired size can be extracted.

3.1.1 Algorithm R

For a sample of size n we need to construct a reservoir of size $\geq n$ be sequentially reading the records, so that at any given point, the reservoir can be used to sample n elements representative of the full data. To avoid expensive post-processing once the data is read, efficient algorithms maintain a set of *candidates* that constitute the true random sample of size n.

1. store the first n records into the reservoir
2. with every following record r: randomly decide whether to keep record r in the reservoir or not
3. if keeping r: randomly select an element s from the set of candidates that will be replaced by r
4. continue until all records are processed

[8]https://www.postgresql.org/ftp/source/v10.5/

Notes. In point 2, we want every element to have the same probability of ending up in the sample, regardless of when it was read. If we want a true random sample of size n selected from R records, every record $r \in R$ is selected to the sample with probability $\frac{n}{R}$. This means that when reading the $t+1$st record, this record should be included in the reservoir with probability $\geq \frac{n}{t+1}$. To allow this, the reservoir needs to be of a minimal size

$$n + \sum_{n \leq t < N} \frac{n}{t+1} = n(1 + H_N = H_n) \approx n(1 + ln\ \frac{N}{n}),$$

where H_k is the kth harmonic number, i.e., $H_k = \sum_{i=1}^{k} \frac{1}{i}$.

There are a number of ways how to speed this procedure up (and Jeff talks about those in his paper), mostly by avoiding the na ive implementation and saving some computational effort, but the structure remains the same. For example, one way to speed things up is not to decide about each record separately, but rather deciding how many records will be skipped (not included in the reservoir) until the next one to be included.

3.2 Size of the sample.

The other important question is how big a sample we need in order to construct reasonably precise histogram. [**?**] introduced an algorithm and a metric to construct approximate histograms over data of any (and unknown) distribution and this has appeared for the first time in SQL Server 7.0; we don't know if the same or similar techniques are still in use nowadays. Postgres is using the bounds stated in this paper.
The idea behind histogram sampling is as follows:

1. select a random sample R of size r from your data
   (data = set of values V of cardinality n)

2. construct an equi-height histogram for R
   that is, find separators $s_i, i \in \{1, \ldots, k-1\}$, so that all the buckets B_j, $B_j = \{v_i \in V | s_{j-1} < v_i \leq s_j\}$[9] are of the same cardinality (or close enough)

3. use the separators s_i from the previous step as separators of the full histogram

Error metric. The metric for histogram error is based around the notion of *δ-separation.* Two histograms with k buckets each are said to be δ-separated if the pair-wise symmetric difference of their corresponding buckets is of size at most δ.

What is the intuitive meaning of δ and its reasonable size? The optimal bucket size for a histogram over n values with k buckets is $\frac{n}{k}$. Consider rewriting δ as $f \times \frac{n}{k}$ for a relative deviation $f \in (0, 1)$ from the optimal k-histogram.

[9]We put $s_0 = -\infty$ and $s_k = +\infty$ for sanity.

Sampling Size. Let $\delta = f \times \frac{n}{k}, f \in (0,1)$ and $\gamma > 0$. An equi-height k-histogram for a random sample R of size r from a value set V of size n gives a δ-deviant k-histogram for V with probability at least $1 - \gamma$, provided that

$$r \geq \frac{4k \ln(2n/\gamma)}{f^2}.$$

Observe that (1) the sample size required grows linearly in the number of buckets k and inversely with the squared deviation δ, but is almost independent of the size of the data n (since the logarithm dependence is negligible), and (2) to get higher confidence γ that the random sample provides the desired error bound requires little additional sampling.

3.3 Adaptive Sampling via Cross-Validation

To retrieve, read and process one record, one needs to load a full block of data (e.g., a page). However, it may not be optimal to read r pages just to provide r records for the sampling and histogram construction. [**?**] offers a different sampling strategy, which also considers optimising the number of record-blocks/pages loaded. The proposed algorithm suggests computing the number of random samples on-the-fly, with termination based on a stopping rule which guarantees convergence to the desired error bound.

The idea is to sample disc blocks (g_i) iteratively and to use the newest sample to cross-validate the accuracy of the histogram built so far, and if the validation fails the we use the sample to update the histogram.

1. Compute r and $g_i = \frac{r}{b}$, $i = 0$ from n, f, k, γ and b.
   (b denotes the number of records in each disk block)
2. Pick a random sample R consisting of g_i disk blocks.
3. Using R, create an equi-height histogram H_0.
4. repeat
   (a) $i := i + 1$
   (b) Pick a new random sample R_i with $g_i = 2^{i-1} g_0$ disk blocks, compute the max error δ_i in partitioning R_i using the separators of H_{i-1}.
   (c) Merge R_i with R. Build a histogram H_i from R.
5. until $\delta_i \leq f \cdot \frac{g_i}{k}$.
6. output current histogram H_i.

Comment In [?] the authors claim the problem of distinct value estimation to be a hard and relatively unsolved problem, with only few analytic results available. [?] extends the claim to "distinct values cannot be approximated reliably".

Appendices

A Postgres – statistics

Postgres collects table/relation statistics when a command `ANALYZE` is called; the detailed statistics are gathered in `pg_statistics` table and a human-readable relevant information composes `pg_stats` table. The contents are documented[10] and reproduced in Tables 1 and 2.

[10]https://www.postgresql.org/docs/10/view-pg-stats.html

name	type	description
`schemaname`	`name`	Name of schema containing table
`tablename`	`name`	Name of table
`attname`	`name`	Name of the column described by this row
`null_frac`	`real`	Fraction of column entries that are `null`
`n_distinct`	`real`	> 0: the estimated number of distinct values in the column < 0: 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.)[11]
`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`	`real[]`	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.)
`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`	`real[]`	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`	`real[]`	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.)

Table 1: Relevant contents of `pg_stats` table.

name	type	description
`inherited`	`bool`	`True`: the row includes inheritance child columns, not just the values in the specified table
`avg_width`	`integer`	Average width in bytes of column's entries
`correlation`	`real`	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.)

Table 2: Not so relevant contents of `pg_stats` table.

B SQL Server – histogram steps and sampling

The following is taken straight from an answer by Joe Obbish on StackExchange (13/04/2017)[12], I copied it here because I find it quite informative and understandable. If at any point anyone needs to know what it is *precisely* that SQL Server is doing, this is a good place to start.

How are the number of Histogram steps decided in Statistics

When SQL Server determines that a statistics update is needed it kicks off a hidden query that reads either all of a table's data or a sample of the table's data. You can view these queries with extended events. There is a function called StatMan within SQL Server that is involved with creating the histograms. For simple statistics objects there are at least two different types of StatMan queries (there are different queries for quick stat updates and I suspect that the incremental stats feature on partitioned tables also uses a different query).

The first one just grabs all of the data from the table without any filtering. You can see this when the table is very small or you gather stats with the `FULLSCAN` option:

```
CREATE TABLE X_SHOW_ME_STATMAN (N INT);
CREATE STATISTICS X_STAT_X_SHOW_ME_STATMAN ON X_SHOW_ME_STATMAN
   (N);
```

[12]https://dba.stackexchange.com/questions/159790/how-are-the-number-of-histogram-steps-decided-in-statistics/159875

```
-- after gathering stats with 1 row in table
SELECT StatMan([SC0]) FROM
(
    SELECT TOP 100 PERCENT [N] AS [SC0]
    FROM [dbo].[X_SHOW_ME_STATMAN] WITH (READUNCOMMITTED)
    ORDER BY [SC0]
) AS _MS_UPDSTATS_TBL
OPTION (MAXDOP 16);
```

SQL Server picks the automatic sample size based on the size of the table (I think that it's both number of rows and pages in the table). If a table is too big then the automatic sample size falls below 100%. Here's what I got for the same table with 1M rows:

```
-- after gathering stats with 1 M rows in table
SELECT StatMan([SC0], [SB0000]) FROM
(
    SELECT TOP 100 PERCENT [SC0], step_direction([SC0]) over (order
       by NULL) AS [SB0000]
    FROM
    (
        SELECT [N] AS [SC0]
        FROM [dbo].[X_SHOW_ME_STATMAN] TABLESAMPLE SYSTEM
           (6.666667e+001 PERCENT) WITH (READUNCOMMITTED)
    ) AS _MS_UPDSTATS_TBL_HELPER
    ORDER BY [SC0], [SB0000]
) AS _MS_UPDSTATS_TBL
OPTION (MAXDOP 1);
```

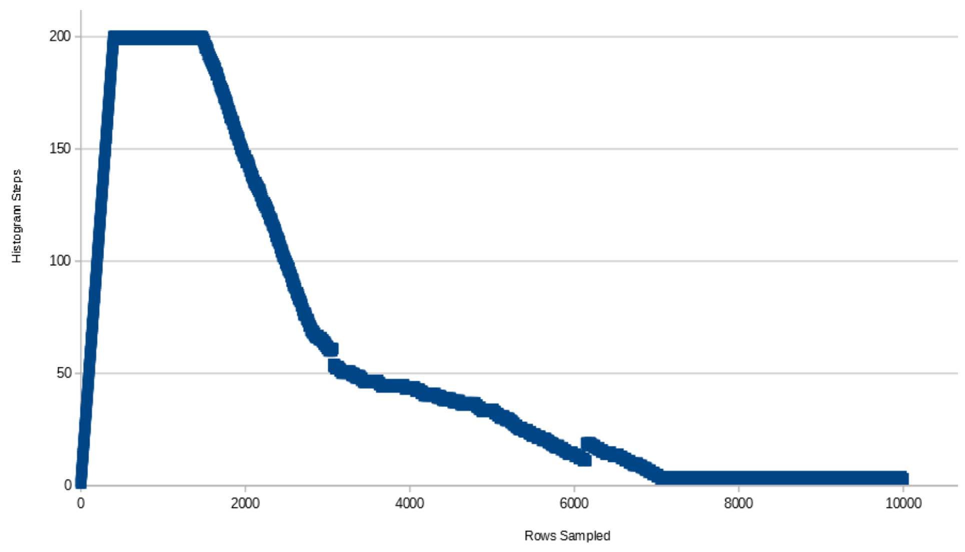
`TABLESAMPLE` is documented but `StatMan` and `step_direction` are not. here SQL Server samples around 66.6% of the data from the table to create the histogram. What this means is that you could get a different number of histogram steps when updating `stats` (without `FULLSCAN`) on the same data. I've never observed this in practice but I don't see why it wouldn't be possible.

Let's run a few tests on simple data to see how the `stats` change over time. Below is some test code that I wrote to insert sequential integers into a table, gather `stats` after each insert, and save information about the stats into a results table. Let's start with just inserting 1 row at a time up to 10000. Test bed:

```
DECLARE
@stats_id INT,
@table_object_id INT,
@rows_per_loop INT = 1,
@num_of_loops INT = 10000,
@loop_num INT;

BEGIN
    SET NOCOUNT ON;

    TRUNCATE TABLE X_STATS_RESULTS;

    SET @table_object_id = OBJECT_ID ('X_SEQ_NUM');
    SELECT @stats_id = stats_id FROM sys.stats
    WHERE OBJECT_ID = @table_object_id
    AND name = 'X_STATS_SEQ_INT_FULL';

    SET @loop_num = 0;
    WHILE @loop_num < @num_of_loops
    BEGIN
        SET @loop_num = @loop_num + 1;

        INSERT INTO X_SEQ_NUM WITH (TABLOCK)
        SELECT @rows_per_loop * (@loop_num - 1) + N FROM
            dbo.GetNums(@rows_per_loop);

        UPDATE STATISTICS X_SEQ_NUM X_STATS_SEQ_INT_FULL WITH
            FULLSCAN;
            -- can comment out FULLSCAN as needed

        INSERT INTO X_STATS_RESULTS WITH (TABLOCK)
        SELECT 'X_STATS_SEQ_INT_FULL', @rows_per_loop * @loop_num,
            rows_sampled, steps
        FROM sys.dm_db_stats_properties(@table_object_id,
            @stats_id);
        END;
END;
```

For this data the number of histogram steps quickly increases to 200 (it first hits the max number of steps with 397 rows), stays at 199 or 200 until 1485 rows are in the table, then slowly decreases until the histogram only has 3 or 4 steps. Here's a graph of all of the data:

Here's the the histogram looks like for 10k rows:



RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_ RANGE_ROWS	AVG_ RANGE_ROWS
1	0	1	0	1
9999	9997	1	9997	1
10000	0	1	0	1

Is it a problem that the histogram only has 3 steps? It looks like information is preserved from our point of view. Note that because the datatype is an `INTEGER` we can figure out how many rows are in the table for each integer from 1 - 10000. Typically SQL Server can figure this out too, although there are some cases in which this doesn't quite work out. See this SE post for an example of this.

What do you think will happen if we delete a single row from the table and update `stats`? Ideally we'd get another histogram step to show that the missing integer is no longer in the table.

```
DELETE FROM X_SEQ_NUM
WHERE X_NUM = 1000;

UPDATE STATISTICS X_SEQ_NUM X_STATS_SEQ_INT_FULL WITH FULLSCAN;

DBCC SHOW_STATISTICS ('X_SEQ_NUM', 'X_STATS_SEQ_INT_FULL');
  -- still 3 steps

DELETE FROM X_SEQ_NUM
WHERE X_NUM IN (2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000);

UPDATE STATISTICS X_SEQ_NUM X_STATS_SEQ_INT_FULL WITH FULLSCAN;
```

```
DBCC SHOW_STATISTICS ('X_SEQ_NUM', 'X_STATS_SEQ_INT_FULL');
   -- still 3 steps
```

That's a little disappointing. If we were building a histogram by hand we would add a step for each missing value. SQL Server is using a general purpose algorithm so for some data sets we may be able to come up with a more suitable histogram than the code that it uses. Of course, the practical difference between getting 0 or 1 row from a table is very small. I get the same results when testing with 20000 rows which each integer having 2 rows in the table. The histogram does not gain steps as I delete data.

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_ RANGE_ROWS	AVG_ RANGE_ROWS
1	0	2	0	1
9999	19994	2	9997	2
10000	0	2	0	1

If I test with 1 million rows with each integer having 100 rows in the table I get slightly better results, but I can still construct a better histogram by hand.

```
truncate table X_SEQ_NUM;

BEGIN TRANSACTION;
INSERT INTO X_SEQ_NUM WITH (TABLOCK)
SELECT N FROM dbo.GetNums(10000);
GO 100
COMMIT TRANSACTION;

UPDATE STATISTICS X_SEQ_NUM X_STATS_SEQ_INT_FULL WITH FULLSCAN;

DBCC SHOW_STATISTICS ('X_SEQ_NUM', 'X_STATS_SEQ_INT_FULL');
   -- 4 steps

DELETE FROM X_SEQ_NUM
WHERE X_NUM = 1000;

UPDATE STATISTICS X_SEQ_NUM X_STATS_SEQ_INT_FULL WITH FULLSCAN;

DBCC SHOW_STATISTICS ('X_SEQ_NUM', 'X_STATS_SEQ_INT_FULL');
   -- now 5 steps with a RANGE_HI_KEY of 998 (?)

DELETE FROM X_SEQ_NUM
WHERE X_NUM IN (2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000);

UPDATE STATISTICS X_SEQ_NUM X_STATS_SEQ_INT_FULL WITH FULLSCAN;
```

```
DBCC SHOW_STATISTICS ('X_SEQ_NUM', 'X_STATS_SEQ_INT_FULL');
   -- still 5 steps
```

Final histogram:

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_ RANGE_ROWS	AVG_ RANGE_ROWS
1	0	100	0	1
998	99600	100	996	100
3983	298100	100	2981	100
9999	600900	100	6009	100
10000	0	100	0	1

Let's test further with sequential integers but with more rows in the table. Note that for tables that are too small manually specifying a sample size will have no effect, so I will add 100 rows in each insert and gather `stats` each time up to 1 million rows. I see (Fig. B) a similar pattern as before, except once I get to 637300 rows in the table I no longer sample 100% of the rows in the table with the default sample rate. As I gain rows the number of histogram steps increases. Perhaps this is because SQL Server ends up with more gaps in the data as the number of unsampled rows in the table increases. I do not hit 200 steps even at 1 M rows, but if I kept adding rows I expect I would get there and eventually start going back down.

Now let's do some simple tests with `VARCHAR` data.

```
CREATE TABLE X_SEQ_STR (X_STR VARCHAR(5));
CREATE STATISTICS X_SEQ_STR ON X_SEQ_STR(X_STR);
   -- Here I am inserting 200 numbers (as strings) along with NULL.

INSERT INTO X_SEQ_STR
SELECT N FROM dbo.GetNums(200)
UNION ALL
SELECT NULL;

UPDATE STATISTICS X_SEQ_STR X_SEQ_STR ;

DBCC SHOW_STATISTICS ('X_SEQ_STR', 'X_SEQ_STR');
   -- 111 steps, RANGE_ROWS is 0 or 1 for all steps
```

Note that `NULL` always gets its own histogram step when it is found in the table. SQL Server could have given me exactly 201 steps to preserve all information but it did not do that. Technically information is lost because '1111' sorts between '1'

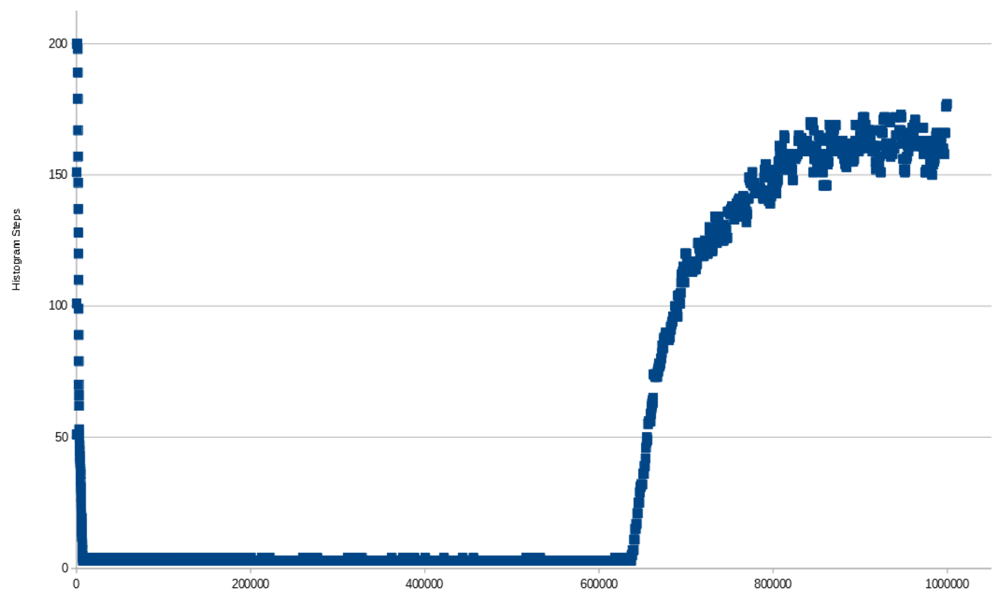


Figure 1: The X-axis is the number of rows in the table. As the number of rows increases the rows sampled varies a bit and doesn't go over 650k.

and '2' for example.

Now let's try inserting different characters instead of just integers:

```
truncate table X_SEQ_STR;

INSERT INTO X_SEQ_STR
SELECT CHAR(10 + N) FROM dbo.GetNums(200)
UNION ALL
SELECT NULL;

UPDATE STATISTICS X_SEQ_STR X_SEQ_STR ;

DBCC SHOW_STATISTICS ('X_SEQ_STR', 'X_SEQ_STR');
   -- 95 steps, RANGE_ROWS is 0 or 1 or 2
```

No real difference from the last test.

Now let's try inserting characters but putting different numbers of each character in the table. For example, `CHAR(11)` has 1 row, `CHAR(12)` has 2 rows, etc.

```
truncate table X_SEQ_STR;

DECLARE
```

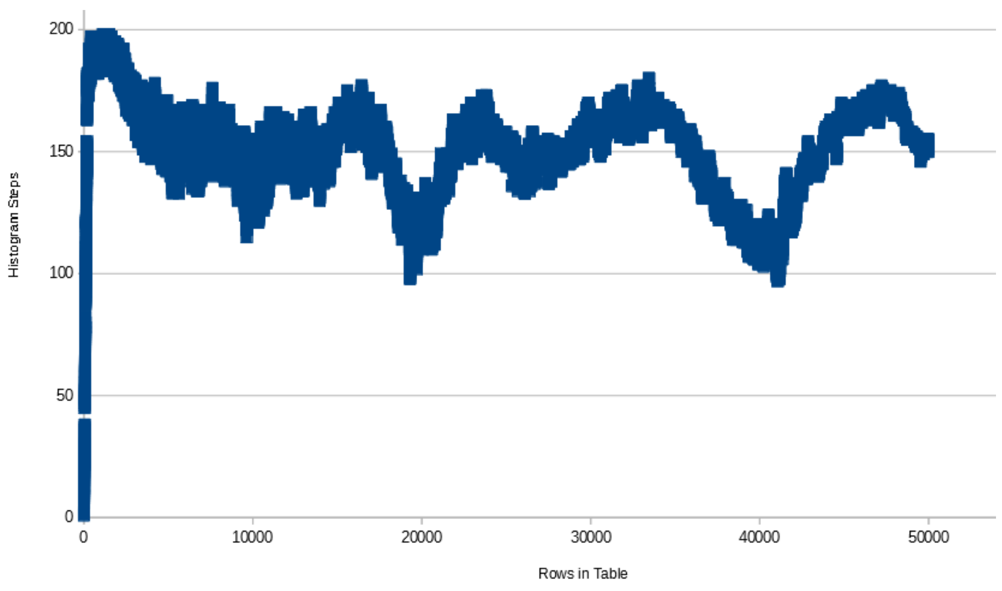
```
@loop_num INT;

BEGIN
    SET NOCOUNT ON;

    SET @loop_num = 0;
    WHILE @loop_num < 200
    BEGIN
        SET @loop_num = @loop_num + 1;

        INSERT INTO X_SEQ_STR WITH (TABLOCK)
        SELECT CHAR(10 + @loop_num) FROM dbo.GetNums(@loop_num);
    END;
END;

UPDATE STATISTICS X_SEQ_STR X_SEQ_STR ;

DBCC SHOW_STATISTICS ('X_SEQ_STR', 'X_SEQ_STR');
   -- 148 steps, most with RANGE_ROWS of 0
```

As before I still don't get exactly 200 histogram steps. However, many of the steps have `RANGE_ROWS` of 0.

For the final test, I'm going to insert a random string of 5 characters in each loop and gather `stats` each time. Here's the code the random string:

```
  char((rand()*25 + 65)) + char((rand()*25 + 65)) +
+ char((rand()*25 + 65)) + char((rand()*25 + 65)) +
+ char((rand()*25 + 65))
```

Here is the graph of rows in table vs histogram steps:



Note that the number of steps doesn't dip below 100 once it starts going up and down. I've heard from somewhere (but can't source it right now) that the SQL Server histogram building algorithm combines histogram steps as it runs out of room for them. So you can end up with drastic changes in the number of steps just by adding a little data. Here's one sample of the data that I found interesting:

ROWS_IN_TABLE	ROWS_SAMPLED	STEPS
36661	36661	133
36662	36662	143
36663	36663	143
36664	36664	141
36665	36665	138

Even when sampling with `FULLSCAN`, adding a single row can increase the number of steps by 10, keep it constant, then decrease it by 2, then decrease it by 3.

What can we summarize from all of this? I can't prove any of this, but these observations appear to hold true:

- SQL Server uses a general use algorithm to create the histograms. For some data distributions it may be possible to create a more complete representation of the data by hand.
- If there is `NULL` data in the table and the `stats` query finds it then that `NULL` data always gets its own histogram step.
- The minimum value found in the table gets its own histogram step with `RANGE_ROWS` = 0.

- The maximum value found in the table will be the final `RANGE_HI_KEY` in the table.
- As SQL Server samples more data it may need to combine existing steps to make room for the new data that it finds. If you look at enough histograms you may see common values repeat for `DISTINCT_RANGE_ROWS` or `RANGE_ROWS`. For example, 255 shows up a bunch of times for `RANGE_ROWS` and `DISTINCT_RANGE_ROWS` for the final test case here.
- For simple data distributions you may see SQL Server combine sequential data into one histogram step that causes no loss of information. However when adding gaps to the data the histogram may not adjust in the way you would hope. When is all of this a problem? It's a problem when a query performs poorly due to a histogram that is unable to represent the data distribution in a way for the query optimizer to make good decisions. I think there's a tendency to think that having more histogram steps is always better and for there to be consternation when SQL Server generates a histogram on millions of rows or more but doesn't use exactly 200 or 201 histogram steps. However, I have seen plenty of stats problems even when the histogram has 200 or 201 steps. We don't have any control over how many histogram steps that SQL Server generates for a statistics object so I wouldn't worry about it. However, there are some steps that you can take when you experience poor performing queries caused by `stats` issues. I will give an extremely brief overview.
  - Gathering statistics in full can help in some cases. For very large tables the auto sample size may be less than 1% of the rows in the table. Sometimes that can lead to bad plans depending on the data disruption in the column. Microsofts's documentation for `CREATE STATISTICS` and `UPDATE STATISTICS` says as much:
    `SAMPLE` is useful for special cases in which the query plan, based on default sampling, is not optimal. In most situations, it is not necessary to specify `SAMPLE` because the query optimizer already uses sampling and determines the statistically significant sample size by default, as required to create high-quality query plans.
  - For most workloads, a full scan is not required, and default sampling is adequate. However, certain workloads that are sensitive to widely varying data distributions may require an increased sample size, or even a full scan.
  - In some cases creating filtered statistics can help. You may have a column with skewed data and many different distinct values. If there are certain values in the data that are commonly filtered on you can create a statistics histogram for just those common values. The query optimizer can use the statistics defined on a smaller range of data

instead of the statistics defined on all column values. You still are not guaranteed to get 200 steps in the histogram, but if you create the filtered stats on just one value you will a histogram step that value.

- Using a partitioned view is one way to effectively get more than 200 steps for a table. Suppose that you can easily split up a large table into one table per year. You create a `UNION ALL` view that combines all of the yearly tables. Each table will have its own histogram. Note that the new incremental statistics introduced in SQL Server 2014 only allows `stats` updates to be more efficient. The query optimizer will not use the statistics that are created per partition.

There are many more tests that could be run here, so I encourage you to experiment. I did this testing on SQL Server 2014 express so really there's nothing stopping you.