Exploiting Ordered Dictionaries to Efficiently Construct Histograms with Q-Error Guarantees in SAP HANA

Guido Moerkotte[#] David DeHaan[◇] Norman May[†] Anisoara Nica[◇] Alexander Boehm[†]

[#]University of Mannheim
Mannheim, Germany
moer@db.informatik.uni-mannheim.de

[◇]SAP AG
Waterloo, Canada
{dave.dehaan,anisoara.nica}@sap.com

[†]SAP AG
Walldorf, Germany
{norman.may,alexander.boehm}@sap.com

ABSTRACT

Histograms that guarantee a maximum multiplicative error (q-error) for estimates may significantly improve the plan quality of query optimizers. However, the construction time for histograms with maximum q-error was too high for practical use cases. In this paper we extend this concept with a threshold, i.e., an estimate or true cardinality θ, below which we do not care about the q-error because we still expect optimal plans. This allows us to develop far more efficient construction algorithms for histograms with bounded error. The test for θ,q-acceptability developed also exploits the order-preserving dictionary encoding of SAP HANA. We have integrated this family of histograms into SAP HANA, and we report on the construction time, histograms size, and estimation errors on real-world data sets. In virtually all cases the histograms can be constructed in far less than one second, requiring less than 5% of space compared to the original compressed data.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*

Keywords

Histogram, Cardinality Estimation

1. INTRODUCTION

Everyone implementing a database management system knows the following embarrassing situation. A complaint drops in that a certain query is too slow. A brief analysis of the situation reveals that the plan for the query is far from optimal and that this is due to cardinality estimation errors.

In order to prevent these embarrassing situations, the management of SAP HANA gave us the task to build histograms that are

1. space efficient,
2. fast to construct, and
3. precise.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2595629.

Obviously, these directives are vague. After a few discussions, it turned out that space efficient means that the histogram may not require more than 10% of the space the compressed column the histogram is built for takes. Further, fast to construct means that construction time is less than a second.

The meaning of precise is more complex. In the literature, many different measures are proposed to assess the precision of histograms (reviewed in [2]). Note that we do not need *any* measure of precision but one that is tightly bound to plan quality. Among the various techniques proposed so far, this requirement is only met by the q-error [1, 2, 6, 9, 13] as has been shown in [13]. However, the q-error has a small problem. If f is the true cardinality and $\hat{f}$ is an estimate, the q-error of $\hat{f}$ is defined as $\max(f/\hat{f}, \hat{f}/f)$. Thus, for a very selective query which returns only a single tuple ($f = 1$) and where the estimate is $\hat{f} = 10$, the q-error is already 10, which is considered unbearable [13]. On the one hand, preventing large q-errors in these cases leads to large histograms. On the other hand, for these small cardinalities, we do not really care, as the effect on the quality of the generated query plan is typically rather limited. In order to capture this intuition, we introduce a new measure of precision called θ,q-acceptability (Sec. 3).

This notion of precision immediately introduces a series of challenges. As always, our histograms consist of buckets. For a given bucket, we must ensure that it is θ,q-acceptable. Thus, we need efficient tests for it (Sec. 4). For the q-error, we have the following: if every estimate for every bucket is precise, i.e., obeys a maximal q-error, the whole histogram is precise. This is no longer true for θ,q-acceptability. Thus, we need means to carry over θ,q-acceptability from single buckets to whole histograms. This is done via some non-trivial theorems presented in Sec. 5. Further, we need space efficient representations of buckets, which are also computationally efficient during estimation (Sec. 6). Last, we need efficient algorithms to construct θ,q-acceptable histograms (Sec. 7). Finally, we performed an evaluation on real datasets in order to assess whether we met the requirements imposed on us by the management (Sec. 8). Of course, this will not be the end of the road (Sec. 10).

The work presented here was carried out in the context of SAP HANA [4]. We especially exploit its ordered dictionaries to fulfill the above-mentioned requirements. Finally, let us emphasize that the techniques are implemented in SAP HANA and will be part of the next release.

2. PRELIMINARIES

In this section, we provide some basic background on the assumed system architecture for the histograms presented. We also introduce some notation and survey basic properties of the q-error.

2.1 Ordered Dictionaries

Modern column stores like SAP HANA [4], SQL Server Apollo [10], or IBM BLU [16] encode the values of one column in a dictionary. The dictionary maps a possibly variable-length domain value to a fixed-size dictionary entry. This encoding reduces memory consumption if the dictionary value can be represented with fewer bits than the original value, and if these values are referenced multiple times. Additionally, the fixed-size representation of values is advantageous for modern CPU architectures which can process dictionary-encoded values with efficient SIMD instructions. Moreover, if the dictionary encoding preserves the original ordering of the domain values, queries can be evaluated efficiently on the encoded data. Finally, the dictionary encoding for the read-optimized storage results in a dense domain of values where the largest possible value is known from the dictionary. Summarizing, SAP HANA uses an order-preserving dictionary with dense encoding on the read-optimized storage [4]. We exploit this fact for the efficient construction of precise and concise histograms.

The experiments in [9] show for three commercial database systems that the q-error was often larger than 1000. When running those experiments initially with SAP HANA, we got similar results. The fact that cardinality estimates may result in unbounded errors is not acceptable because the resulting query execution plans may be far from the optimal plan. This motivated us to investigate how we could integrate the Q-optimal histograms presented in [9] into the SAP HANA database. In this paper, we explain how the dictionary encoding can be exploited to construct histograms faster than before while at the same time consuming less space.

2.2 Notation

For the rest of the paper, let R be a relation and A one of its attributes. The ordered domain of A is denoted by $\mathcal{D}$. The distinct values occurring in A are denoted by $x_1, \ldots, x_d$, i.e., $\Pi^D_A(R) = \{x_1, \ldots, x_d\}$. Without loss of generality, we assume that $x_1 < \ldots < x_d$. We use f_i to denote the frequency of value x_i in attribute A. Thus, $\{(x_i, f_i)\}$ is the attribute density.

An *ordered dictionary* maps $\{x_1, \ldots, x_d\}$ onto some dense integer domain, e.g. $[0, d[$ in an order-preserving way. That is, the order on $\{x_1, \ldots, x_d\}$ is compatible with the order on integers. The elements in the dense integer domain are called *dictionary identifiers*. Then, these dictionary identifiers are stored within the attributes instead of the original values. The consequence is that the attribute's domain becomes discrete (integers) and dense, i.e., there is no value in its domain without an occurrence in the attribute.

We concentrate on range queries of the form

$$\sigma_{c_1 \leq A < c_2}(R),$$

and we are interested in its result cardinality and estimates thereof. Other forms of range queries and exact match queries can easily be translated into this form. For brevity, we identify a range query with a simple interval $[c_1, c_2[$. Then, for a given range query $[c_1, c_2[$, the cardinality of its result is the *accumulated frequency* $f^+(c_1, c_2) = \sum_{c_1 \leq x_i < c_2} f_i$.

A histogram is a sequence of buckets B_i $(1 \leq i \leq \beta)$. Each bucket B_i is described by bucket boundaries b_{i-1}, b_i comprising an interval $[b_{i-1}, b_i[$. Each bucket contains some additional information in the form of some numbers. These numbers are coefficients for a binary estimation function $\hat{f}^+$ and the goal is that $\hat{f}^+(c_1, c_2)$ be close to $f^+(c_1, c_2)$. In Sec. 2.3, we define what we mean by close. In Sec. 2.4, we show that if we demand certain mathematical properties for our estimation function, then there is not much of a choice for it.

2.3 Q-Error

Let $f \geq 0$ be a number and $\hat{f} \geq 0$ be an estimate for f. Then, the *q-error* of the estimate $\hat{f}$ is defined as [1, 2, 6, 9, 13]:

$$||\hat{f}/f||_Q,$$

where $||x||_Q := \max(x, 1/x)$.

If for some value $q \geq 1$ $||\hat{f}/f||_Q \leq q$, we say that the estimate is *q-acceptable*.

Let $\hat{f}^+(x, y)$ be an estimation function for f^+ on the interval $[l, u[$. Let $q \geq 1$ be some number. We say that $\hat{f}^+$ is q-acceptable on $[l, u[$, if for all $l \leq c_1 \leq c_2 \leq u$ the estimate $\hat{f}^+(c_1, c_2)$ is *q-acceptable* on $[c_1, c_2[$.

The q-error has some nice properties, which we summarize here. For $1 \leq i \leq n$, let f_i be true values and $\hat{f}_i$ be estimates with $||\hat{f}_i/f_i||_Q \leq q$ for all $1 \leq i \leq n$. Then

$$1/q * \sum_{i=1}^{n} f_i \leq \sum_{i=1}^{n} \hat{f}_i \leq q * \sum_{i=1}^{n} f_i$$

holds, i.e.,

$$||\frac{\sum_{i=1}^{n} \hat{f}_i}{\sum_{i=1}^{n} f_i}||_Q \leq q$$

and also for $||\hat{f}_i/f_i||_Q \leq q_i$ for all $1 \leq i \leq n$

$$\prod_{i=1}^{n}(1/q_i) * \prod_{i=1}^{n} f_i \leq \prod_{i=1}^{n} \hat{f}_i \leq \prod_{i=1}^{n} q_i * \prod_{i=1}^{n} f_i$$

holds, i.e.,

$$||\frac{\prod_{i=1}^{n} \hat{f}_i}{\prod_{i=1}^{n} f_i}||_Q \leq \prod_{i=1}^{n} q_i.$$

2.4 Properties of (Estimation) Functions

Since we consider range predicates, we need binary estimation functions $\hat{f}^+(x, y)$ such that for a given range query $[c_1, c_2[$ the estimation function $\hat{f}^+$ generates an estimate $\hat{f}^+(c_1, c_2) \geq 0$. Since we build histograms, these estimation functions are defined piecewise. Estimation functions may share some properties with other functions. Let us briefly review them.

An estimation function $\hat{f}^+$ is called *monotonic* on $[l, u]$, if and only if for all $l \leq c_1 \leq c_1' \leq c_2' \leq c_2 \leq u$

$$\hat{f}^+(c_1', c_2') \leq \hat{f}^+(c_1, c_2)$$

holds.

An estimation function $\hat{f}^+$ is called *additive* on $[l, u]$, if and only if for all $l = c_1 \leq \ldots \leq c_k = u$

$$\hat{f}^+(c_1, c_k) = \sum_{i=1}^{k-1} \hat{f}^+(c_i, c_{i+1})$$

holds. Note that every additive estimation function is monotonic.

Assume that we have an additive linear estimation function

$$\hat{f}^+(x, y) = \alpha x + \beta y + \gamma.$$

for arbitrary coefficients α, β, and γ. Then, it follows that for all x, y, z with $x \leq y \leq z$

$$\begin{aligned} & \alpha x + \beta z + \gamma = (\alpha x + \beta y + \gamma) + (\alpha y + \beta z + \gamma) \\ \prec\succ \quad & 0 = \alpha y + \beta y + \gamma \end{aligned}$$

This can only be achieved if $\gamma = 0$ and $\alpha = -\beta$. Thus, every linear and additive estimation function is of the form

$$\hat{f}^+(x, y) = \alpha(y - x).$$

We typically wish the bucket's estimation function to be precise for the whole bucket. Thus, we demand that

$$||\hat{f}^+(\mathrm{l},\mathrm{u})/f^+(\mathrm{l},\mathrm{u})||_Q \leq q$$

for some error bound q. With $f^+ := f^+(\mathrm{l},\mathrm{u})$, we have

$$\begin{array}{lll}(1/q)f^+ \leq & \hat{f}^+(\mathrm{l},\mathrm{u}) & \leq qf^+ \\ (1/q)f^+ \leq & \alpha(\mathrm{u}-\mathrm{l}) & \leq qf^+\end{array}$$

and thus

$$(1/q)\frac{f^+}{\mathrm{u}-\mathrm{l}} \leq \alpha \leq q\frac{f^+}{\mathrm{u}-\mathrm{l}}. \quad (1)$$

The above holds if

$$||\frac{\alpha}{\frac{f^+}{\mathrm{u}-\mathrm{l}}}||_Q = ||\frac{\alpha(\mathrm{u}-\mathrm{l})}{f^+}||_Q \leq q. \quad (2)$$

This clearly holds if we use the usual estimation function

$$\hat{f}^+_{\text{avg}}(x,y) = \frac{y-x}{\mathrm{u}-\mathrm{l}}f^+ = \alpha(y-x) \quad (3)$$

for $\alpha = f^+/(\mathrm{u}-\mathrm{l})$. $\hat{f}^+_{\text{avg}}$ is the estimation function of choice for our histograms. Alternatively, we have the possibility to choose α within certain bounds given by Eq. 1.

3. θ,Q-ACCEPTABILITY

One problem occurs if the cardinality estimate for some query is $\hat{f} \geq 1$ and the true cardinality is zero. This happens, since we never want to return an estimate of zero, because this leads to query simplifications which may be wrong or in reorderings which may not be appropriate. To solve this dilemma, consider the following thought experiment: During query optimization time, we execute building blocks and even access paths until the first tuple has been delivered. From there on, we know for sure whether the result will be empty or not. If there is a tuple delivered, we buffer it, since we want to avoid its recalculation at runtime. The overhead of this method should therefore be low. Now, assume that we are willing to buffer more tuples (say 1000). Then, if there are less than 1000 qualifying tuples, we know the exact answer after fetching them. If we have to halt the evaluation of the build block since the buffer is full, we know that there will be ≥ 1000 qualifying tuples. Let us denote by θ_{buf} the number of tuples we are willing to buffer. This way of interleaving query optimization and query execution can be considered a small step in the direction of *adaptive query optimization* [3]. A more comprehensive analysis can be found in [15].

However, before we can evaluate a building block or access paths, we have to determine an optimal one, which in turn requires cardinality estimates! This brings us to the concept of θ,q-acceptability. In essence, its idea is that cardinality estimates may be imprecise as long as they do not badly influence the decisions of the query optimizer. This means that as long as the query optimizer produces the best plan, any estimate is o.k. Let us for example take the decision whether to exploit an index or not. Assume that an index is better than a scan if less than 10% of the tuples qualify (This is a typical value [7, 11]). If the relation has 10000 tuples, the threshold is at 1000 tuples. Thus, assume that for a given range query both the estimate and the true value do not exceed 500. Then, no matter what the estimate is, we should use the index. Note that the q-error can be 500 (e.g., the estimate is 1 and the true value is 500). Still it does not have any bad influence on our decision. The important thing is that the estimate has to be precise around 1000. For a given relation and one of its indices, we denote by θ_{idx} the number of tuples that, if exceeded, make a table scan more efficient than the index scan.

Let us now combine these two aspects. Assume we want to have a maximal q-error of q. Define $\theta = \min(\theta_{\text{buf}} - 1, (1/q)\theta_{\text{idx}})$. Assume that $\hat{f}$ is an estimate for the true cardinality f. Further assume that if $\hat{f}$ or f exceeds θ, then $||\hat{f}/f||_Q \leq q$. Now let us go through the optimizer. In a first step, we define our building blocks and access paths, which requires to decide on index usage. Clearly, the estimate will be precise above $(1/q)\theta_{\text{idx}}$, which includes the critical part. After evaluating a building block or access path, we have precise cardinality estimates if fewer than θ_{buf} tuples are retrieved. Otherwise, our estimate will obey the given q-error. Thus, we are as precise as necessary under all circumstances.

These simple observations motivate us to introduce the notion of θ, q-acceptability. Let $f \geq 0$ be a number and $\hat{f} \geq 0$ be an estimate for f. Let $q \geq 1$ and $\theta \geq 1$ be numbers. We say that $\hat{f}$ is θ, q-acceptable if

1. $f \leq \theta \wedge \hat{f} \leq \theta$ or
2. $||\hat{f}/f||_Q \leq q$.

We define that a binary estimation $\hat{f}^+$ for f^+ is θ, q-acceptable on some interval $[l,u]$, if for all $l \leq c_1 \leq c_2 \leq u$ the estimate $\hat{f}^+(c_1,c_2)$ is θ, q-acceptable.

Another way to look at θ is that if the cardinality is below it, we do not really care how large it really is. The reason is that query execution for relations smaller than θ will be fast anyway, even if we pick the wrong plan. As the HANA database does not implement the idea of buffering, we consider θ as the number of rows below which any plan will be close to optimal.

4. TESTING θ,Q-ACCEPTABILITY

We assume that our range queries are of the form $[c_1, c_2[$. For discrete domains, we can convert any range and exact match query to this form. During histogram construction we need to reject buckets which are not θ,q acceptable. Thus, we need an efficient test for θ,q acceptability. This is the topic of this section.

4.1 Discretization

Directly testing θ,q-acceptability for a given bucket for a continuous domain is impossible since it would involve testing θ,q-acceptability of $\hat{f}^+(c_1,c_2)$ for all c_1, c_2 within the bucket. In this subsection, we show that a test quadratic in the number of distinct values in the bucket suffices.

Let $[c_1, c_2[$ be a query interval. Assume i, j are chosen such that $[x_i, x_j] \subseteq [c_1, c_2] \subset [x_{i-1}, x_{j+1}]$. Since there is no distinct value between x_i and x_{i-1} and between x_j and x_{j+1}, we have that $f^+(c_1,c_2) = f^+(x_i,x_j) < f^+(x_{i-1},x_{j+1})$. Assume the following conditions hold:

1. $\hat{f}^+$ is monotonic.
2. $||\frac{\hat{f}^+(x_i,x_j)}{f^+(c_1,c_2)}||_Q \leq q$
3. $||\frac{\hat{f}^+(x_{i-1},x_{j+1})}{f^+(c_1,c_2)}||_Q \leq q$

Since $\hat{f}^+(x_i,x_j) = \hat{f}^+(c_1,c_2) \leq \hat{f}^+(x_{i-1},x_{j+1})$, we then have $||\frac{\hat{f}^+(c_1,c_2)}{f^+(c_1,c_2)}||_Q \leq q$.

Exploiting this fact, we can develop the following quadratic test for some given θ and q.

THEOREM 4.1. If for all i, j such that x_i and x_j are in the bucket, we have that

$$\hat{f}^+(x_{i-1},x_{j+1}) \leq \theta \wedge f^+(x_{i-1},x_{j+1}) \leq \theta$$

or

$$||\frac{\hat{f}^+(x_i,x_j)}{f^+(x_i,x_j)}||_Q \leq q \ \wedge \ ||\frac{\hat{f}^+(x_{i-1},x_{j+1})}{f^+(x_i,x_j)}||_Q \leq q,$$

then the estimation function is θ,q-acceptable for the bucket. □

4.2 The Sub-quadratic Test

Still, after discretization, the number of tests is quadratic in the number of distinct values contained in a bucket. We can restrict this even further for monotonic and additive estimators $\hat{f}^+$. For a given, fixed θ and for any i ($1 \leq i < d$), we define i' to be the index such that

1. $f^+(x_i, x_{i'}) \leq \theta$,
2. $\hat{f}^+(x_i, x_{i'}) \leq \theta$ and
3. $f^+(x_i, x_{i'+1}) > \theta$ or $\hat{f}^+(x_i, x_{i'+1}) > \theta$

This index i' can be found by binary search.

For a given L, assume that for all l with $1 \leq l \leq L$

- $||\hat{f}^+(x_i, x_{i'+l})/f^+(x_i, x_{i'+l})||_Q \leq q$,

and for some k, we have

- $f^+(x_i, x_{i'+L}) \geq k\theta$ and
- $\hat{f}^+(x_i, x_{i'+L}) \geq k\theta$.

That is, we stop after L tests as soon as the latter two conditions are satisfied. Then, one can prove the following theorem:

THEOREM 4.2. An estimation function satisfying the above conditions is θ, $(q + \frac{1}{k})$-acceptable. □

Summarizing, we are able to trade in accuracy for performance when testing θ, q-acceptability of a given estimation function.

4.3 A Cheap Pretest for Dense Buckets

If the domain of the attribute is discrete and every domain value within the bucket has a frequency larger than zero, then the bucket is *dense*. This is always the case if ordered dictionaries are used. In this case, we have the following theorem:

THEOREM 4.3. If the bucket is dense and

1. the cumulated bucket frequency is less than or equal to θ or
2. $\max_i f_i / \min_i f_i \leq q^2$,

then there exists an estimation function $\hat{f}^+$ that is θ,q-acceptable. □

The first condition also holds for non-dense buckets. The second condition only holds if we use our flexibility concerning the α in our approximation function (see Eq. 1). If we use $\hat{f}^+_{\text{avg}}$ (see Eq. 3), we need to exchange the second condition by

$$q\overline{f} \geq \max_i f_i \ \wedge \ (1/q)\overline{f} \leq \min_i f_i,$$

where $\overline{f}$ is the average frequency of the bucket.

4.4 A Combined Test

In our histograms, we use $\hat{f}_{\text{avg}}$ as the estimation function. During histogram construction, we need to check whether for some bucket bound by $[l, u[$ our estimation function $\hat{f}_{\text{avg}}$ is θ,q-acceptable. We do so by the function `checkThetaQAcc`. Its simple pseudo code is

```
isThetaQAcc(l, u)
if the pretest of Sec. 4.3 succeeds
        then return true
if MaxSize < (l − u)
        then return false
return the result of the sub-quadratic test of Sec. 4.2
```

where `MaxSize` is set to 300 in our experiments. The reason is that even the sub-quadratic test can be quite expensive, if bucket sizes get too large. Thus, if the pretest cannot determine θ,q-acceptability, we restrict the application of the sub-quadratic test to buckets which are not too large.

4.5 Bounding θ, q-Violation Size

Given an estimator $\hat{f}^+$, we call a range query $[x, y[$ a θ, q-*violation for* $\hat{f}^+$ if $\hat{f}^+(x, y)$ is not θ, q-acceptable. We call a θ, q-violation *minimal* if it does not strictly contain another θ, q-violation. In order to prove that an estimator is θ, q-acceptable, it clearly suffices to prove that there do not exist any minimal θ, q-violations.

THEOREM 4.4. Let $[x_i, x_j[$ be a $0, q$-violation for $\hat{f}^+$, and let $x_{i'}$ be any value satisfying $x_i < x_{i'} < x_j$. Then at most one of the estimates $\hat{f}^+(x_i, x_{i'})$ and $\hat{f}^+(x_{i'}, x_j)$ is $0, q$-acceptable. □

COROLLARY 4.1. If $[x_i, x_j[$ is a minimal $0, q$-violation for $\hat{f}^+$, then $j = i + 1$. □

Corollary 4.1 implies that θ, q-acceptability of $\hat{f}^+$ can be tested in linear time when $\theta = 0$ because it suffices to only examine ranges containing a single distinct value. The following theorem generalizes this special case.

THEOREM 4.5. Let $[x_i, x_j[$ be a θ, q-violation for $\hat{f}^+_{\text{avg}}$, and let $x_{i'}$ be any value satisfying $x_i < x_{i'} < x_j$. If

1. $f^+(x_i, x_{i'}) > \theta$ or $\hat{f}^+_{\text{avg}}(x_i, x_{i'}) > \theta$ and
2. $f^+(x_{i'}, x_j) > \theta$ or $\hat{f}^+_{\text{avg}}(x_{i'}, x_j) > \theta$,

then θ, q-violation $[x_i, x_j[$ is not minimal. □

COROLLARY 4.2. Given a dense bucket of n values, if $[x_i, x_j[$ is a minimal θ, q-violation for $\hat{f}^+_{\text{avg}}$, then $j - i < \frac{2\theta n}{f^+} + 3$. □

Corollary 4.2 implies that the expense of testing θ, q-acceptability of $\hat{f}^+_{\text{avg}}$ for dense buckets is proportional to the ratio between θ and the cumulated bucket frequency, and can be tested in $O(\frac{\theta}{f^+}n^2 + n)$ time. (Note that θ, q-acceptability is trivial when $f^+ \leq \theta$, and that for dense buckets $f^+ \geq n$ always holds.) A similar result can be derived for non-dense buckets by bounding the maximum width $|x_j - x_i|$ of the minimal θ, q-violation $[x_i, x_j[$.

4.6 Dynamically Adjusting θ

The previous section showed that θ, q-acceptability can be decided in time proportional to θ. Unfortunately, pressure to minimize histogram size motivates large choices for θ. A strategy for partially mitigating the effect of large θ values is suggested by the following axiom, which follows directly from the definition of θ, q-acceptability.

AXIOM 4.1. θ', q-acceptability of $\hat{f}^+$ implies θ, q-acceptability of $\hat{f}^+$ for all $\theta' < \theta$. □

```
isThetaQAccDynamic(l, u)
θ' := 0
for j := l + 1 up to u
        for i := j − 1 down to max(l, j − ⌈2θ'(u−l)/f+⌉ − 3)
            if f̂+(i, j) is not θ, q-acceptable
                then    θ' := max(f+(i, j), f̂+(i, j))
                        if θ' > θ then return false
return true
```

Figure 1: Testing θ, q-acceptability with dynamic θ

For any $\theta' < \theta$, testing θ', q-acceptability can be used as a cheap acceptance filter for testing θ, q-acceptability. This suggests that for cases where θ, q-acceptability holds, an iterative approach of testing θ', q-acceptability for $\theta' \ll \theta$ and then iteratively increasing θ' if needed could be more efficient than testing θ, q-acceptability directly. Rather than iterating entire invocations of the decision procedure, it is an improvement to increase the value of θ' dynamically within a single invocation.

Decision procedure `checkThetaQAccDynamic` (see Fig. 1) enumerates query intervals by advancing the right endpoint j and then considering all left endpoints i within a search length bounded by results from Sec. 4.5. Each θ', q-violation forces an increase to θ' which also increases the search length. In practice, the algorithm includes heuristic pretests as in Sec. 4.4, which have been omitted here, and can be extended to handle non-dense buckets.

4.7 Exploiting Recent History

Consider algorithm `checkThetaQAccDynamic` in Fig. 1. Suppose a θ', q-violation $[i', j'[$ with $j' \ll u$ causes θ' to be increased to a large value $\leq \theta$. For all future iterations $j > j'$, the search length will continue to depend upon the large value θ' even though no value within $[i', j'[$ will be examined again once $j - j'$ exceeds the search length (assuming that no further θ', q-violations are found). The following theorem and corollaries permit extensions to `checkThetaQAccDynamic` that exploit knowledge of recently-enumerated intervals.

THEOREM 4.6. Let $[x_i, x_j[$ be a minimal θ, q-violation for $\hat{f}^+$, and let $x_{i'}$ be any value satisfying $x_i < x_{i'} < x_j$.

1. If $\hat{f}^+(x_i, x_{i'})$ is $0, q$-acceptable, then $f^+(x_{i'}, x_j) \leq \theta$ and $\hat{f}^+(x_{i'}, x_j) \leq \theta$.
2. If $\hat{f}^+(x_{i'}, x_j)$ is $0, q$-acceptable, then $f^+(x_i, x_{i'}) \leq \theta$ and $\hat{f}^+(x_i, x_{i'}) \leq \theta$.

□

COROLLARY 4.3. Given a dense bucket of n values, if $[x_i, x_j[$ is a minimal θ, q-violation for $\hat{f}^+_{\text{avg}}$ and $x_{i'}$ is the largest value satisfying $x_i < x_{i'} < x_j$ such that $[x_{i'}, x_j[$ is $0, q$-acceptable, then $j - i < \frac{\theta n}{f^+} + (j - i') + 1$. □

COROLLARY 4.4. Let $[x_i, x_j[$ be a minimal θ, q-violation for $\hat{f}^+$. If $[x_{j-1}, x_j[$ is $0, q$-acceptable, then $[x_i, x_{j-1}[$ is a $0, q$-violation but not a θ, q-violation. □

Corollary 4.3 permits the search length for iteration j to be recomputed after the first $0, q$-acceptable estimate $\hat{f}^+(i, j)$ is encountered. Corollary 4.4 permits the entire backward search for iteration j to be skipped if $\hat{f}^+(j-1, j)$ is $0, q$-acceptable and the previous iteration $j - 1$ did not find any $0, q$-violations for $\hat{f}^+$ with right endpoint at $j - 1$.

5. FROM BUCKETS TO HISTOGRAMS

In general, θ,q-acceptability does not carry over from buckets to histograms. Consider a histogram in which each bucket has the true cumulated frequency θ and the estimate for each bucket is 1. It follows that each bucket is θ,q-acceptable. Further, the estimate for a range query comprising n buckets is n and the true value is $n\theta$ and, thus, the q-error of the estimate is θ. Clearly, the histogram is not θ,q-acceptable if $q < \theta$. However, with some effort, one can prove the following theorems, which give us upper bounds for the q-error of the whole histogram. These theorems are at the core of our approach, since they allow us to restrict testing for the θ,q-acceptability of the whole histogram to merely testing θ,q-acceptability for single buckets.

We start with two consecutive buckets.

THEOREM 5.1. *Let H be a histogram. Consider two neighbored buckets B_1 and B_2 spanning the intervals $[b_i, b_{i+1}[$ for $i = 0, 1$. Let $k \geq 2$ be a number. If both buckets B_1 and B_2 are θ,q-acceptable, then the histogram is $k\theta$,$q + \frac{q}{k-1}$-acceptable.*

Now, we extend this to a sequence of buckets.

THEOREM 5.2. *Let H be a histogram. Consider $n \geq 3$ consecutive buckets B_i in H spanning the intervals $[b_i, b_{i+1}[$ for $i = 0, \ldots, n$. Let $k \geq 3$ be a number. If every estimate for a range query spanning a whole bucket is q-acceptable and every bucket B_i is θ,q-acceptable, then the histogram is $k\theta$,$q + \frac{2q}{k-2}$-acceptable.*

In case the estimates for a whole bucket are precise, e.g., if we use $\hat{f}^+_{\text{avg}}$, we can refine the bounds.

COROLLARY 5.3. *Let H be a histogram. Consider $n \geq 3$ consecutive buckets B_i in H spanning the intervals $[b_i, b_{i+1}[$ for $i = 0, \ldots, n$. Let $k \geq 3$ be a number. If every estimate for a range query spanning a whole bucket is 1-acceptable and every bucket B_i is θ,q-acceptable, then the histogram is $k\theta$,q'-acceptable, where $q' := \frac{2}{k-2}q + 1$.*

Note that we now have two θs and qs. One pair of θ, q for buckets and one pair θ', q' for the whole histogram. Our histogram construction algorithms will take care that the buckets are θ,q-acceptable. We will call these inner θ and inner q. Then, using the theorems above we can derive upper bounds for q' given some θ' for the whole histogram. However, the true maximal value of q' given some θ' that can be observed experimentally may be lower than the upper bound derived in the theorems. Thus, we determine these maximal occurring values for q' experimentally in Sec. 8.

6. BUCKET CONTENTS

Any synopsis is a lossy compression of the true attribute density. Thus, it might not come as a surprise that we apply compression techniques for numbers. The first one is the well-known q-compression [5, 9, 14], which we review in Sec. 6.1.1. Then, we introduce binary q-compression (Sec. 6.1.2), which has the advantage of being faster to calculate. Next, Sec. 6.2 shows how we can exploit these compression techniques to build a whole bunch of interesting bucket types.

6.1 Q-Compression

6.1.1 General Q-Compression

The goal of q-compression is to approximate a number $x \geq 1$ with a small q-error. Given some number $b > 0$, let x be some number in the interval $[b^{2l}, b^{2(l+1)}]$. If we approximate x by b^{2l+1},

qcompressb(x, b)
return (0 == x) ? 0 : $\lceil log_b(x) \rceil + 1$

qdecompressb(y, b)
return (0 == y) ? 0 : $b^{y-1+0.5}$

qcompressbase(x, k)
// x is the largest number to be compressed
// k is the number of bits used to store a compressed value
return $x^{1/((1<<k)-1)}$

Figure 2: Q-compression, $\log_b$-based

#Bits	Base	Largest compressible number	q-error
4	2.5	372529	1.58
4	2.6	645099	1.61
4	2.7	1094189	1.64
5	1.7	8193465	1.30
5	1.8	45517159	1.34
5	1.9	230466617	1.38
6	1.2	81140	1.10
6	1.3	11600797	1.14
6	1.4	1147990282	1.18
7	1.1	164239	1.05
7	1.2	9480625727	1.10
8	1.1	32639389743	1.05

Table 1: Examples for q-compression

then $||b^{2l+1}/x||_Q \leq b$. Let $x_{\max}$ be the largest number to be compressed. If $x_{\max} \leq b^{2(k+1)}$ for some k is the maximal occurring number, we can approximate any x in $[1, x_{\max}]$ with $\lceil \log_2(k) \rceil$ bits obeying a maximal q-error of b. Note that in our context we know the largest value after we have generated the dictionary during the delta merge. We can extend q-compression to allow for the compression of 0, as in the code in Fig. 2. There, we use the base b instead of b^2 as above. Thus, the error is at most $\sqrt{b}$. Let us consider a concrete example. Let $b = 1.1$. Assume we use 8 bits to store a number. Then, since $1.1^{254} \approx 32.6 * 10^9$, we can approximate even huge numbers with a small q-error of at most $\sqrt{1.1} = 1.0488$. Other examples are given in Table 1.

There exists a small disadvantage of q-compression with a general base. Though calculating the logarithm is quite cheap, calculating the power during decompression is quite expensive. On the hardware used in our experiments, compression takes roughly 67 ns, whereas decompression takes 168 ns. This is bad since in the context of cardinality estimation, decompression is used far more often than compression. Thus, we introduce an alternative called *binary q-compression*.

6.1.2 Binary Q-Compression

The idea of binary q-compression is simple. Let x be the number we want to compress. If we take the base $b = 2$, then $\lceil \log_2(x) \rceil = k$, where k is the index of the highest bit set. This calculation can be done by a rather efficient machine instruction. This gives us a maximum q-error of $\sqrt{2}$. We can go below this by remembering not only the highest bit set, but the k highest bits set. Additionally, we store the position of them (their shift) in s bits. The pseudo code is given in Fig. 3, where we extended the scheme to allow for the compression of zero. So far, this resembles a special floating point representation with only positive mantissa and exponent. We now apply a small trick to increase efficiency.

k	max q-error observed	max q-error theoretical ($\sqrt{1+2^{1-k}}$)
1	1.5	1.41
2	1.25	1.22
3	1.13	1.12
4	1.07	1.06
5	1.036	1.03
6	1.018	1.016
7	1.0091	1.0078
8	1.0045	1.0039
9	1.0023	1.00195
10	1.0011	1.00098
11	1.00056	1.00048
12	1.00027	1.00024

Table 2: Maximum observed vs. maximum theoretical q-error

qcompress2(x, k, s)
if $2^s > x$
then
  bits = x
  shift = 0
else
  shift = index-of-highest-bit-set(x) - k + 1;
  bits = (x >> shift)
return (bits << shift) | shift

qdecompress2(y, k, s)
shift = y & $(2^s - 1)$
bits = y >> shift
x = bits << shift
// assume C = (int) ((sqrt((double) 2.0) - 1.0) * 4 * (1 << 30))
x |= (C >> (32 - shift))
return x

Figure 3: Binary Q-compression

The q-middle of 2^n and $2^{n-1} - 1$ is $\sqrt{2^n * (2^{n+1} - 1)}$. This is the estimate we should return. However, we do not want to compute the square root during decompression, since this is too expensive. A little calculation helps.

$$\begin{aligned} \sqrt{2^n * (2^{n+1} - 1)} &\approx \sqrt{2^n * 2^{n+1}} \\ &= \sqrt{2^{2n} * 2} \\ &= \sqrt{2} * 2^n \\ &= 2^n + (\sqrt{2} - 1) * 2^n \end{aligned}$$

The second part can be calculated by a constant $(\sqrt{2}-1)$ shifted by n to the left. The pseudo code in Fig. 3 gives the calculation of this constant C in C. The best theoretical q-error achievable with storing k bits is $\sqrt{1+2^{1-k}}$. With our fast approximation, we get pretty close, as Table 2 shows. The observed maximal q-error column was obtained experimentally. The deviation from the observed maximal q-error to the theoretical maximal q-error is due to the fact that only a small portion of the digits of C are used. Further, compression (3.4 ns) and decompression (5.0 ns) are fast.

6.1.3 Incremental Updates

It might come as a surprise that q-compressed numbers can be incrementally updated. Already in 1978, Morris observed this fact [14]. Later, Flajolet analyzed the probabilistic counting method thoroughly [5]. The main idea is rather simple. For binary q-

	Total		Bucklets			
Name	#Bits	Compression	#	#Bits	Compression	Base
QC16T8x6	16	bq-compr.	8	6	q-compr.	1.2-1.4
QC8x8	0	–	8	8	q-compr.	1.1
QC16x4	0	–	16	4	q-compr.	2.5-2.7
QC8T8x7	8	q-compr.	8	7	q-compr.	1.1-1.2
BQC8x8	0	–	8	8	bq-compr.	–

Table 3: Simple bucket types

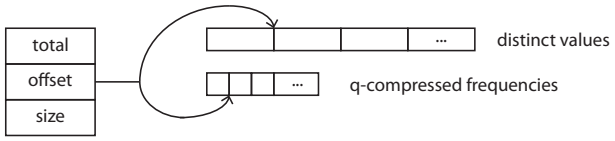

Figure 4: Illustration of QRawNonDense bucket

compressed numbers, the randomized increment function is defined as follows:

`RandomIncrement`(int& c)
// c: the counter
let δ be a binary random variable which takes
    value 1 with probability 2^{-c} and
    value 0 with probability $1 - 2^{-c}$.
c += δ

To go to an arbitrary base, we have to modify the random variable δ such that it takes the value 1 with probability a^{-c} and 0 with probability $1 - a^{-c}$.

6.2 Bucket Types

Using q-compression or binary q-compression, we can store several cumulated frequencies in 64 bits and possibly add a total to make estimates for query ranges which span many buckets cheaper. We will make use of this fact in our histogram construction algorithm by subdividing a bucket into several smaller bucklets, where we will use eight bucklets per bucket. All the coefficients α (cf. Eq. 3) of all bucklets will then be stored in one unit of 64 bits. Additionally, we may store the total (T) of the whole bucket in compressed form. Of course there are plenty of possibilities to store eight or nine differently compressed numbers in one chunk of 64 bits. Some useful combinations are given in Table 3. As indicated in Table 1, a smaller base leads to a smaller error but the largest number compressible also becomes smaller. Therefore, different bases should be used, as appropriate for the bucket. Which base is actually used can be indicated in the bucket's header. In our histograms, we preferably use the QC16T8x6 bucket because it is an excellent choice (see Table 3). It compresses the total frequency of a bucket into 16 bits using the fast binary q-compression. The cumulated frequencies of the eight bucklets are stored in 6 bits, each using q-compression. Consider a histogram consisting solely of QC16T8x6 buckets and a query range that wholly contains n consecutive buckets and partially touches two neighboring buckets at each side. Then, in the worst case, the estimation time is $5.0n+16*168$ ns on our hardware. Note that using q-compression adds an additional small error to the error bounds derived in the theorems in Sec. 5. But as shown experimentally in Sec. 8, these small additional errors do not matter.

Some attribute distributions contain parts which are not approximable. For these parts, we need two additional bucket types. The first bucket type is for dense buckets, i.e., the attribute's domain is discrete and all possible values between the bucket boundaries have a frequency larger than zero. In this case, it suffices to store the frequencies of each distinct value contained in the bucket. We use q-compression with 4 bits to store the frequencies. This results in a `QCRawDense` bucket. In case the bucket is non-dense, we additionally need to store the distinct values. In this case, we have two arrays, one with the distinct values and the other one consisting of 4 bits per entry to store the q-compressed frequencies. Then, we have a `QCRawNonDense` bucket. This bucket type is illustrated in Fig. 4. The 64 bit header consists of a 32 bit offset into two aligned arrays. One contains the distinct values, the other the q-compressed frequencies. Additionally, `size` (16 bit) determines the number of distinct values and `total` (16 bit) contains the binary q-compressed cumulated frequency.

7. HISTOGRAM CONSTRUCTION

We present two construction algorithms for homogeneous histograms, which contain buckets of a single type. Both divide buckets into eight bucklets. The first alternative divides a bucket into equi-width bucklets and supports any estimation function, the second into variable-width bucklets working only with $\hat{f}^{+}_{\text{avg}}$.

7.1 Equi-Width Bucklets

In this kind of histogram, which we call *QEWH*, we divide each bucket into eight equi-width bucklets. Since we have a discrete and dense domain (due to ordered dictionaries), every bucklet within a bucket will contain the same number m of distinct values. We then use the QC16T8x6 bucket to store the cumulated frequency of the bucket, i.e., the total, and the cumulated frequencies of each bucklet. As usual, we need to store the bucket boundaries. In order to have error guarantees for our histograms, we have to make sure that each bucklet is θ,q-acceptable. Then, the theorems of Sec. 5 together with the error guarantees of q-compression give us an overall error guarantee for the whole histogram.

Histogram construction is performed by two functions. The main function (`BuildQEWH`) constructs buckets, encodes them into the QC16T8x6 bucket and appends them to the histogram. In order to find the upper bound of a bucket, it calls `FindLargest`. Given a start l for a bucket, it finds the maximal number of distinct values m for each bucklet such that the bucket is still θ,q-acceptable for some given θ and q. Thereby, it makes use of the θ,q-acceptability test `isThetaQAcc` defined in Sec. 4.4.

Fig. 5 contains the pseudo code of both routines. In order to find the largest number of distinct values that fit into a bucklet while still preserving θ,q-acceptability, `FindLargest` starts with $m = 1$ and doubles m until θ,q-acceptability is violated. Then, we know that if there are m_{good} distinct values in each bucklet, every bucklet will be θ,q-acceptable, whereas m_{bad} elements are too many. We then apply binary search to find the maximal m in between that still assures θ,q-acceptability for each bucklet. The main procedure `BuildQEWH` is rather simple. It starts with the first distinct value 0, which becomes the first bucket boundary b_0, and calls `FindLargest`(0) to find the next bucket boundary b_1 and

```
FindLargest(l)
m := 1
while ∀0 ≤ i < 8 isThetaQAcc(l + i * m, l + i * (m + 1)) do
        m := 2*m
m_good := m/2
m_bad := m
find by binary search the maximal m with
        m_good ≤ m < m_bad and ∀0 ≤ i < 8
        isThetaQAcc(l + i * m, l + i * (m + 1))
return m

BuildQEWH
i := 0
b_0 = 0  // b_i will be the bucket boundaries
while b_i < d do  // d is the number of distinct values
        m := FindLargest(b_i)
        b_{i+1} := b_i + 8 * m
        encode the frequencies in the bucklets and the total
            into a QC16T8x6 bucket and append it to the histogram
```

Figure 5: Pseudo code for building histograms with equi-width bucklets

so on. During the course, it constructs a QC16T8x6 representation for each bucket.

7.2 Variable-Width Bucklets

In *QEWH* histograms, each 64-bit QC16T8x6 histogram bucket represents a separate eight-bucklet equi-width histogram over the bucket interval. Unfortunately, if there is a single interval that must be approximated by a narrow bucklet, then all eight bucklets will be narrow, causing poor compression due to a short bucket interval. One solution (not implemented in `BuildQEWH`) suggested in Sec. 6.2 is to use heterogeneous bucket types. Another alternative is to not use equi-width bucklets.

Because the start and end boundaries of each bucket are stored separately, encoding only seven internal bucklet boundaries enables eight variable-width bucklets. We encode the internal boundaries using seven 9-bit integers representing bucklet widths, and one flag bit indicating whether the bucklet widths are measured from the bucket start or end. By combining these 64 bits of boundary information with a QC16T8x6 encoding of the bucklet frequencies, we obtain a new 128-bit bucket type we denote as QC16T8x6+1F7x9. This encoding limits seven of the bucklets to a maximum width of 511, while either the first or last bucklet width is not limited. We call the resulting histogram type *QVWH*.

7.2.1 Incremental Construction

Fig. 6 contains pseudo code for constructing *QVWH* histograms. Function `BuildQVWH` constructs QC16T8x6+1F7x9 buckets by computing the length of each bucklet with a successive call to function `GrowBucklet`, which takes as input the starting position and the maximum permitted length of the next bucklet.

Function `GrowBucklet` could be implemented by modifying `FindLargest` to build a single bucklet, using binary search to enumerate candidate lengths and an independent θ, q-acceptability test for each. This *generate-and-test* methodology is required for building multiple equi-width bucklets, but for a single bucklet we can improve efficiency by eliminating the stratification between choosing bucklet length and testing θ, q-acceptability.

Observe that for the estimator function $\hat{f}^+_{\text{avg}}$ from Eq. 3, the θ, q-acceptability for each query interval depends only on the value of

```
GrowBucklet(l,m_max)
α_LB := 0
α_UB := ∞
for m := 1 up to m_max
        j := l + m
        α := f^+(l,j) / m
        for i := j − 1 down to l
            if f̂^+(i, j) > θ
                then α_LB := max(α_LB, α(j−i) / q)
            α_UB := min(α_UB, max(θ / (j−i), qα(j − i)))
            if α < α_LB or α_UB < α
                then return m − 1
return m_max

BuildQVWH
i := 0
b_0 = 0  // b_i will be the bucket boundaries
while b_i < d do  // d is the number of distinct values
        m_0 := GrowBucklet(b_i, d − b_i)
        b_{i+1} := b_i + m_0
        while ∀1 ≤ j < 8
            if j < 7 or m_0 > 511 then
                m_j := GrowBucklet(b_{i+1}, min(511, d − b_{i+1}))
            else m_j := GrowBucklet(b_{i+1}, d − b_{i+1})
            b_{i+1} := b_{i+1} + m_j
        encode the bucklet frequencies and the widths in m_[0,7]
            into a QC16T8x6+1F7x9 bucket
            and append it to the histogram
```

Figure 6: Pseudo code for building histograms with variable-width bucklets

α. The generate-and-test algorithm revisits the same query intervals for testing each bucklet length to ensure that θ, q-acceptability holds for the newest value of α. However, we can invert this by computing for each query interval a lower and upper bound on valid values for α; then it suffices to enumerate each query interval at most once. Function `GrowBucklet` works by incrementally increasing the bucklet length as long as α stays between lower bound α_{LB} and upper bound α_{UB}. On each step, it enumerates only the query intervals whose right endpoints correspond to the right endpoint of the current bucklet. For each such query, it updates α_{LB} and α_{UB}, if necessary. If after this enumeration $\alpha_{\text{LB}} \leq \alpha \leq \alpha_{\text{UB}}$ does not hold, then a θ, q-violation exists for $\hat{f}^+_{\text{avg}}$, either because the last value added to the bucket caused the value of α to violate a query interval enumerated earlier, or because a query interval containing this last value reduced the valid range that α may take on.

In order to clearly illustrate the incremental construction aspect of `GrowBucklet`, the pseudo code in Fig. 6 has been simplified so that it exhaustively enumerates a quadratic number of query intervals. In practice, all of the techniques in Sec. 4.3-4.7 for reducing the quadratic cost of θ, q-acceptability can be applied to `GrowBucklet` to reduce the number of iterations of its inner loop.

8. EXPERIMENTS

In our experiments, we demonstrate how the histograms presented achieve the goals we stated at the beginning. We focus on two real-world data sets which proved to be even more challenging regarding our goals than the ones presented in [9], i.e. the three com-

mercial systems analyzed in that work would also produce q-errors larger than 1000. Our analysis includes measurements on the time to construct the histograms and the space required for the resulting histograms. Finally, we report on the quality of cardinality estimates obtained with our histograms.

8.1 Experimental Setup

We performed the experiments on a HANA system SPS7 running on a SLES11 SP1 system with Intel Xeon E5-2660 CPUs with 2.2 GHz regular clock speed and 256GB main memory. All binaries were compiled using GCC 4.3.4 with O3. Notice that the construction code is single-threaded, and thus all experiments ran on a single core.

The histograms for each column were created five times, and we report the average construction time for these five runs. For all experiments, we test with a maximal q-error per bucket of 2.0. Hence, we choose a rather small q-error to get high quality estimates. We present detailed experiments on the sensitivity of construction time and space consumption to changes of θ (Sec. 8.5). We also experimented with different values for the q-error. The behavior is rather similar to that of θ and not presented due to a lack of space. If not mentioned otherwise, we let the histogram construction component choose $\lceil \theta = f * \sqrt(|R|) \rceil$ where $f = 0.1$ is configurable, but any other sub-linear function of the cumulated frequency to choose θ would work. In our implementation, we use $\hat{f}_{\text{avg}}$ as the estimation function.

For the plots in this section, we use the following names for histograms tested:

- Bucket type: *1* for atomic (i.e. bucket without bucklets), *F8* for fixed-width eight bucklets, *V8* for variable-width eight bucklets;
- *V* for value-based, *D* for dictionary-encoded and
- Construction method: *gt* for generate-and-test construction (Sec. 7.1), *inc[B]* for incremental construction (Sec. 7.2).

Our implementation of the generate-and-test construction did not include the optimizations discussed in Sec. 4.5-4.7 for bounding the required search length, whereas our implementation of incremental construction included the optimizations from Sec. 4.5-4.7. Because these optimizations are orthogonal to the construction method, to distinguish their effect we also tested the incremental construction with these optimizations disabled. We use *inc* to denote this naïve version of incremental search, and *incB* to denote the incremental search with bounding optimizations.

8.2 Data Sets

The first data set is taken from an internal development system for the SAP ERP. This system is used to analyze mixed OLAP and OLTP workloads. The schema consists of 133 tables with 757 columns in total.

The second data set is derived from customer data of a large warehouse system. The schema consists of 229 tables with 2410 columns in total. The largest of these columns – which proved to be the most challenging one – contains 168 million distinct values.

For most of these columns, it does not make sense to create a histogram: 1) Many columns contained less than 20 distinct values. If needed, we can create a precise statistics for each value for these columns. 2) The value distribution for a primary key or uniqueness constraint is trivial. Notice that we know the number of distinct values from the dictionary created for these columns. Applying these restrictions on the columns resulted in 688 columns of the *ERP* data set and 192 of the *BW* data set.[1]

8.3 Histograms on Values

First, we analyze histograms based on values with a domain that is not necessarily dense. Even though our system provides dictionary encoding, some use cases—such as federation—require histograms based on the raw values. The equi-width and variable-width bucklets of Sec. 7 rely on dense domains, and so for histograms based on values we use atomic buckets. Because the domain is not dense, the distinct value count cannot be computed exactly from the query interval width. As a result, to support estimation of distinct value queries each histogram bucket must store the distinct value count in addition to the cumulated frequency. Because this estimate may contain errors, θ,q-acceptability needs to be tested independently for range and distinct count queries.

In this set of experiments, we analyze two different bucket types for the histograms:

1VincB1 – atomic 16-bit bucket storing cumulated frequency and distinct value count each as 8-bit bq-compressed integers. Built using incremental construction with bounding optimizations, independently testing θ,q-acceptability for both range and distinct count queries.

1VincB2 – identical to 1VincB1 except that the construction algorithm only tests θ,q-acceptability for range queries. Distinct count is stored but permits estimates with unbounded error.

Note that using Corollary 5.3 the error guarantees on the buckets, if present, carry over to θ,q-acceptability for the histograms.

Fig. 7 shows the construction time for the value-based histograms using these two bucket types. The figure was derived by sorting all attributes by their construction time. The x-axis then contains the rank of each attribute in this sorted list and the y-axis represents the construction time in microseconds. As desired, for almost all columns the construction time is below one second for both data sets. Overall, the construction time for the more complex bucket type 1VincB1 exceeds one second for 6 columns ($\approx 1\%$ of the columns): in the BW data set for no column, in the ERP data set for 6 columns, up to 2.6 seconds. We also observe that the construction for 1VincB1 is approximately twice as long as for 1VincB2 buckets because for 1VincB2 the θ,q-acceptability is tested only for range queries, not for distinct value queries.

Fig. 8 shows the memory consumption relative to the original compressed column data for the value-based histograms using these two bucket types. The x-axis contains the rank of the columns but this time after sorting the histograms by their relative memory consumption. The y-axis is the memory consumption relative to the original compressed column data (in percent). For the two tested bucket types more than 10% space is required for a roughly 15% of the columns. Considering that value-based histograms will be used for estimates on remote data, this seems acceptable. Interestingly, the memory consumption is virtually identical for the two algorithms because in almost all cases, the same bucket boundaries were chosen, which indicates that for these data sets frequency estimation is more error-prone than distinct-value estimation.

8.4 Histograms on Dictionary-encoded Values

In this section, we analyze the construction time and space consumption of histograms generated from dictionary-encoded values. Dictionary-encoded values are the default case for histograms in

[1]Note that generated data sets like a generated Zipf distribution or TPC-DS are too simple to approximate.

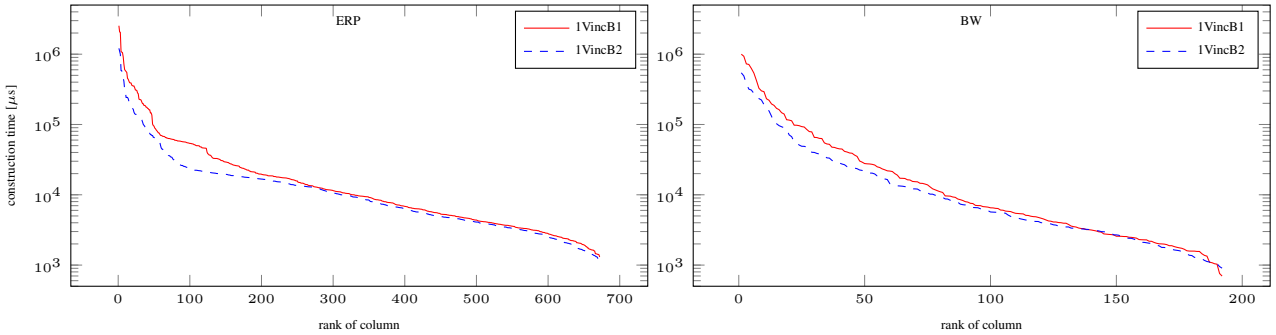

Figure 7: Value-based histograms: histogram construction time [μs], θ chosen by system

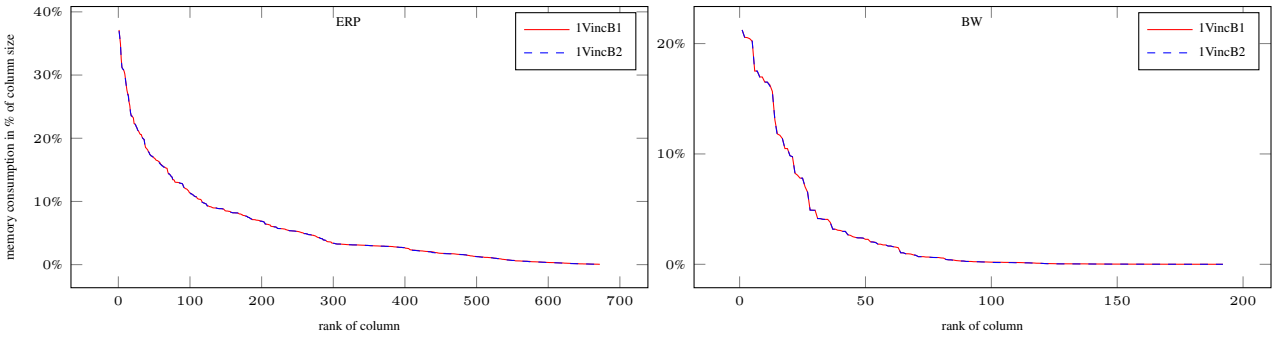

Figure 8: Value-based histograms: memory consumption in percent of original column size, θ chosen by system

SAP HANA. In this section, we analyze histograms made of five different bucket types:

1Dinc – atomic 8-bit bucket storing bq-compressed cumulated frequency. Built using incremental construction without bounding optimizations.

1DincB – same as 1Dinc, but incremental construction uses bounding optimizations.

F8Dgt – 8 fixed-width bucklets in 64-bit QC16T8x6 format. Built using generate-and-test construction.

V8Dinc – 8 variable-width bucklets in 128-bit QC16T8x6+1F7x9 format. Built using incremental construction without bounding optimizations.

V8DincB – same as V8Dinc but incremental construction uses bounding optimizations.

In the figures for this experiment, we also sorted the columns by the construction time and by their relative memory consumption.

Fig. 9 contains the results for construction times for histograms on dictionary-encoded data with fixed-width and variable-width bucklets (and buckets). The plots indicate that the construction of the histograms of about ten columns for each data set requires more than one second. The incremental construction of the variable-width bucklets (V8Dinc) is faster than for the fixed-width bucklets (F8Dgt) for long-running columns. For simple columns (less than 10ms construction time) the construction time for fixed-size bucklets is up to two times faster than the variable-size bucklet. The construction times for the bounded and unbounded incremental search are comparable for simple histograms requiring at most 10ms to construct. For more expensive histograms the incremental algorithms using bounded search are always faster, typically by factors ranging between 1.1 and 2.0.

Fig. 10 contains the results for space consumption of the histogram as percentage of the original column data. Evidently, the space consumption is much better than for the value-based histograms. The highest ratio of memory consumption we observed was about 8% for the most challenging column for the fixed-width histogram. In average, F8Dgt had slightly higher memory consumption than the other bucket types. For all other columns the memory consumption is in the range of 6.5% and well below. Among those, for the incremental algorithms the memory consumption was identical for the bounded and unbounded variants, with the pair V8Dinc[B] having the lowest memory consumption overall. For both the BW data set and the ERP data set the space consumption is below 5% within the top-10 columns, independent of the bucket type. The only exception is the fixed size bucket for the ERP dataset, where 24 columns (out of 688 columns) require more than 5% space of the original column size. But none of these histograms is larger than 1500 bytes.

8.5 Impact of Theta

We analyzed the impact of the parameter θ and the maximum allowed q-error per bucket on both the construction time and memory consumption for the resulting histogram. In this section, we present our results for the BW data set for the incremental construction of variable-width bucklets (V8DincB) for different θ—see Fig. 11. We performed similar analysis for different q-errors.

From the figure it is evident that as we increase the value for θ, the construction time increases while the space consumption decreases. Space consumption is reduced because larger buckets can be constructed that are θ,q-acceptable. Construction time increases because the bound on the worst-case search length is proportional to θ (see Sec. 4.5). For algorithms not including the bounding optimization of Sec. 4.5, worst-case search length is independent of θ and the algorithms illustrate the opposite effect—larger values of θ reduce construction time because the cheap pretest presented in Sec. 4.3 succeeds more often. Increased pretest efficacy for larger θ occurs for all algorithms, but for those that bound search length the effect is masked by the increase in worst-case search length.

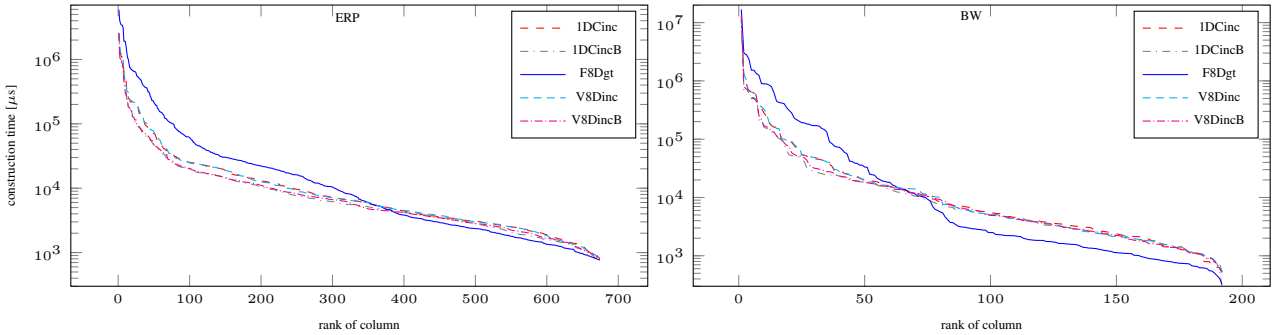

Figure 9: Histograms on dictionary-encoded values: construction time [μs] for different bucket types, θ chosen by system

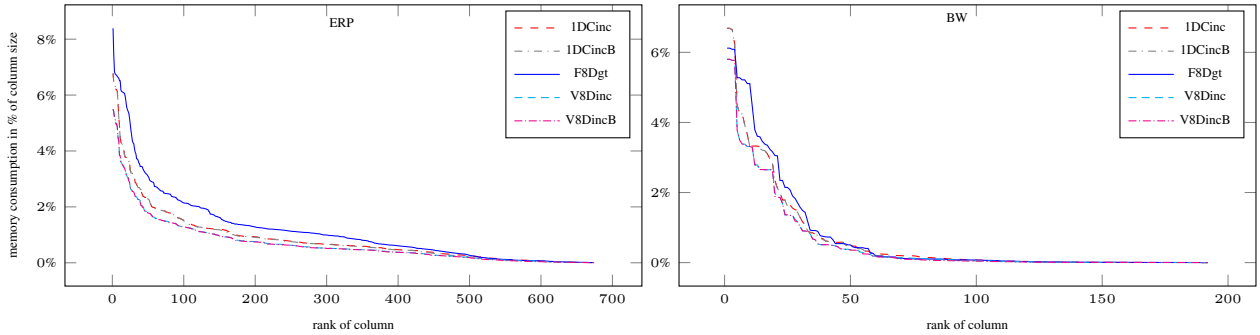

Figure 10: Histograms on dictionary-encoded values: space consumption (%) for different bucket types, θ chosen by system

We can also see that the value for θ chosen by the system is usually in the range we have analyzed. In our concrete measurements on the BW data set the values chosen for θ range from 5 to 1298 (with 626 being the next smaller value).

Increasing the maximum allowed q-error for a bucket tends to reduce the construction time and space consumption. However, we find that achieving a significant reduction in memory consumption requires increasing the maximum allowed q-error by a factor of four or more. We judge this to be a bad trade-off between precision of cardinality estimation and space consumption of the histograms.

8.6 Quality of Estimates

To verify the theorems in Sec. 5, we generated all possible range queries for both data sets. Table 4 presents the results for these tests, where we use $\theta = 32$, $q = 2$ and $k = 1, 2, 3, 4$ and fixed-width buckets (i.e. F8Dgt). As this test ran for several months on our hardware, we do not provide experimental results for other bucket types than F8Dgt. The first two rows provide the information as derived from Corollary 5.3 which holds for all bucket types. The first row shows for $k = 1, 2, 3, 4$ and $\theta = 32$ used for the bucket construction how the θ translates to the whole histogram. The second row gives the values of the maximum theoretical q-error for $k = 3$ and $k = 4$; for $k = 1$ and $k = 2$ the corollary does not provide an error bound.

In the bottom part of the table, we provide the observed errors when running all possible range queries on all columns. The column for $k * \theta = 32$ shows that the corollary is indeed not applicable for $k < 3$, as the observed q-error is as high as 35. For the remaining cases, we never observed the worst-case error predicted by Corollary 5.3. As for F8Dgt, we expect errors below the upper bound of Corollary 5.3 for all other bucket types.

k*θ	32	64	96	128
max. q-error	-	-	5	3
Rank			ERP	
1	35	7.3	2.59	2.51
2	35	7.3	2.58	2.33
3	35	6.6	2.51	2.31
Rank			BW	
1	35	4.9	2.62	2.62
2	30	4.7	2.24	2.23
3	27	4.4	2.22	2.22

Table 4: Top 3 observed maximum q-errors over ERP and BW columns using F8Dgt and ($k = 1, 2, 3, 4$, $q = 2.0$)

9. RELATED WORK

Constructing synopses for cardinality estimates is an intensely studied field, see [2, 8] for good surveys. Most database systems today rely on histograms that cannot guarantee the maximum error returned for cardinality estimates.

For example DB2 BLU [16], the column store of DB2, uses order-preserving dictionaries clustered by the frequency of values. This allows to compress frequent values much more that infrequent values. Additionally, DB2 BLU automatically creates a synopsis table which keeps additional statistics to minimize the number of accessed pages during query processing. For query optimization, DB2 BLU relies on the same equi-depth histograms – based on samples – as the row store of DB2 [8, 16].

SQL Server partitions sets of rows into so called column segments which are independently compressed column-wise [10]. Some of the dictionary encoded values may be shared in a global dictionary. The system either uses dictionary encoding or a value encoding to reduce the number of bits to represent the domain of values. SQL Server creates max-diff histograms from samples of data [8]. As an extension to the SQL Server row store, the samples may be truly random samples of rows instead of samples of pages.

The SAP HANA database encodes all values in order-preserving dictionaries [4]. For the read-optimized storage, these values are encoded using the least number of bits to represent the domain of values of the column. The bit-encoded values are subject to different compression techniques available for the bit-compressed dictio-

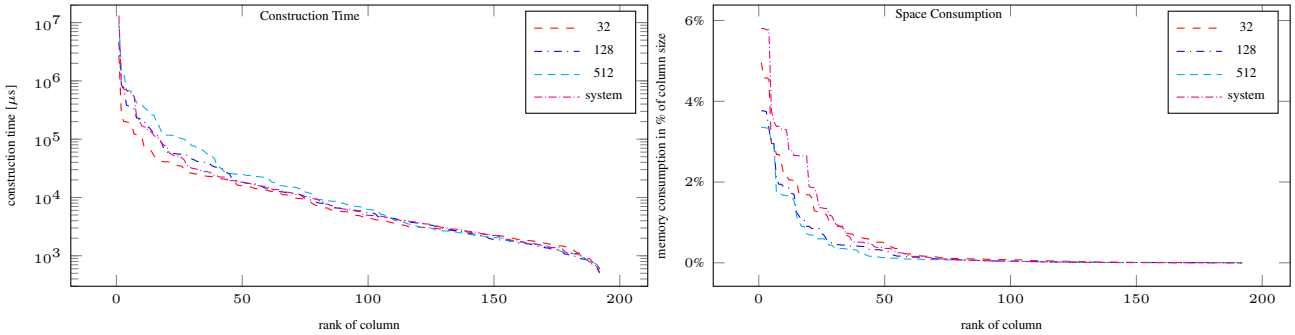

Figure 11: Histogram construction time and space consumption for V8DincB for different θ

nary encoded values. Until today, SAP HANA relied on sampling data as the basis for cardinality estimates. If indexes are available, they are also considered for cardinality estimation.

In contrast to the histograms used in the systems discussed above [9, 13] propose histograms on columns of values which do not need to be dense. Their histograms guarantee a maximum q-error for single-column estimates for range predicates and the number of distinct values. This is a significant improvement because the estimation error propagates with the power of four in the query. Our work extends these results by considering the parameter θ, which allows for faster construction of more compact histograms, while at the same time keeping the estimation error bounded for estimates that have a significant runtime impact. We consider columns on dense domains, as they are common in main-memory column stores to reduce the construction time and space consumption of the histogram. Furthermore, we introduce variable-width bucklets which further reduce the size of the histogram. Like [9], we consider different bucket types, but we currently only consider histograms using a single bucket type. Mixing different bucket types similar to [9] is part of our future work.

An orthogonal approach to deal with imprecise cardinality estimates is presented in [15]. They evaluate a query execution plan to a point where cardinality estimation errors would lead to different plan alternatives. In this case, intermediate results are materialized, and the remaining query is re-optimized based on precise cardinality information derived from the materialized results. In this context, having more precise cardinality estimates to start with reduces the need to reexamine the query execution plan.

10. CONCLUSION

In this paper we have shown how the SAP HANA query optimizer can benefit from histograms that guarantee upper bounds for the estimation error. As novel aspects, we consider the order-preserving dictionary encoding of the SAP HANA column store, which generates a dense encoding of the value domain of a column. As a result, the histograms require less time to construct and consume less space in memory. For practical reasons, we introduced θ,q-acceptability, which only requires a maximum error above a threshold of θ. We presented novel algorithms to construct the buckets and histograms. We refer to [12] for the proofs of the formal aspects of this paper.

Histograms based on buckets using either equi-width or variable-width bucklets lead to low construction times and high-quality estimates. For simple columns the construction time for fixed-size bucklets is smaller than for the variable-size bucklet, but fixed-size bucklets consume less space. Value-based buckets can also be constructed with our algorithms, but they would only be used if the dictionary cannot be used directly, e.g. for data integration.

As the histograms are created for single columns only, the error guarantees only apply for conditions on a single column. Consequently, complex expressions which cover multiple columns including join predicates have to be addressed with conventional techniques. Hence, we need equally precise histograms for two and more dimensions. This is the challenge ahead of us.

11. REFERENCES

[1] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 268–279, 2000.

[2] G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. NOW Press, 2012.

[3] A. Deshpande, Z. Ives, and V. Raman. *Adaptive Query Optimization*. NOW, 2007.

[4] F. Färber et al. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1), 2012.

[5] P. Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, 1985.

[6] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 541–550, 2001.

[7] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4):63–68, 1997.

[8] Y. Ioannidis. The history of histograms (abridged). In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 19–30, 2003.

[9] C.-C. Kanne and G. Moerkotte. Histograms reloaded: the merits of bucket diversity. In *SIGMOD*, pages 663–674, 2010.

[10] P. A. Larson et al. Enhancements to SQL server column stores. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1159–1168, 2013.

[11] N. May, M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Index vs. navigation in XPath evaluation. In *Int. XML Database Symp. (XSym)*, pages 16–30, 2006.

[12] G. Moerkotte. Building query compilers. `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`, 2014.

[13] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2(1):982–993, 2009.

[14] R. Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[15] T. Neumann and C. Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *Proc. der GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft (BTW)*, pages 73–92, 2013.

[16] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 1080–1091, 2013.