Industry and Applications Paper

Fast Entropy Maximization for Selectivity Estimation of Conjunctive Predicates on CPUs and GPUs

Diego Havenstein
SAP SE
diego.havenstein@sap.com

Peter Lysakovski
SAP SE
peter.lysakovski@sap.com

Norman May
SAP SE
norman.may@sap.com

Guido Moerkotte
University of Mannheim
moerkotte@uni-mannheim.de

Gabriele Steidl
University of Kaiserslautern
steidl@mathematik.uni-kl.de

ABSTRACT

Entropy maximization is the only principled approach to combine several (partial) selectivity estimates to an estimate for a full conjunction. However, this approach has no appearance in database management systems. We conjecture that the main reason is a lack of implementations with good performance. Indeed, the originally proposed iterative scaling algorithm has a slow convergence rate and high complexity in each iteration. As an alternative, we propose to use a method based on Newton's algorithm to solve the entropy maximization problem. Further, we show how this general approach can be implemented very efficiently for both CPUs and GPUs. Our experiments show that our CPU and GPU implementation is more than 4 orders of magnitude faster than the state-of-the-art method for the most complex problem it could handle. For even more complex problems our new GPU implementation outperforms our CPU implementation by more than 43x. In a few milliseconds it is now possible to compute all partial selectivities for complex conjunctive predicates with 20 or more predicates. We strongly believe that the proposed implementation is ready for production-grade database management systems.

1 INTRODUCTION

Query optimizers need precise cardinality estimates to generate query execution plans of high quality. Basic approaches to estimate result cardinalities rely on the assumptions that values are uniformly distributed and the selectivities of predicates are independent. Increasingly sophisticated techniques were proposed to address the uniform distribution assumption and also correlation between predicates, see [1, 6] for comprehensive surveys. However, the space consumption and maintenance effort for all combinations of multi-column histograms [12], samples [3], or statistics on views [7] exponentially grows with the number of columns considered. For this reason, these statistics are generated only for a few out of all possible column subsets. We address the challenge how to integrate estimates produced from these sources of statistics consistently. Note that in general sampling alone is not sufficient because it can result in highly imprecise estimates, and thus other synopsis have to be used [11, 15].

Markl et al. [9] observed that the query optimizer makes suboptimal plan choices despite the rich statistics at hand to find the optimal plan because *fleeing to ignorance* seems to be the most reasonable choice. They suggested the maximum entropy method to exploit all available knowledge and to handle inconsistent and missing information in a consistent way. Consider for example the following scenario. Assume we have three predicates p_0, p_1, p_2 whose selectivities are estimated to be $s_0 = 0.5$, $s_1 = 0.5$, and $s_2 = 0.5$. Further assume that the combined selectivity for $p_0 \wedge p_1$ is $s_{01} = 0.4$ and for $p_1 \wedge p_2$ is $s_{12} = 0.1$. These selectivities could be estimates produced from single column histograms, 2-dimensional histograms, and/or sampling. The question is what is the selectivity of the whole conjunct $p_0 \wedge p_1 \wedge p_2$? The answer given by entropy maximization (as proposed by Markl et al. [9]) is 0.08, which clearly deviates from the estimate $0.5 * 0.5 * 0.5 = 0.125$ produced under the independence assumption. Clearly, the estimate produced under the independence assumption is inconsistent since it is larger than the selectivity of $p_1 \wedge p_2$. Indeed, it is widely known that the independence assumption (1) does not hold in general and (2) leads to bad cardinality estimates and, consequently, (3) leads to suboptimal query execution plans [8].

In order to derive the missing selectivity values, Markl et al. propose to find the unique vector $x = (x_0, x_1, \ldots x_{2^z-1})$ (for z predicates) that maximizes the entropy

$$H(s) = \sum_i -x_i \log x_i$$

subject to the constraints given by the known selectivities. A formal definition requires some preliminaries and will be given in Sec. 2. Maximizing entropy can be seen as a generalization of the independence assumption limited to the case of unknown selectivities. Since the known selectivities are possibly derived from several synopsis, the problem may become inconsistent. In this case, a *corrector step* is necessary. Since different correctors have been proposed in the literature (e.g., [9, 10]), we assume in this paper that the problem on hand is consistent.

To solve the entropy maximization problem, Markl et al. use *iterative scaling*. However, this algorithm is known to have very slow convergence [2, p82] and, additionally, has a relatively high asymptotic complexity of $O(m^2 * n)$ in each iteration, where m is the number of known selectivities, z the number of predicates and $n = 2^z$. For example, for eight predicates, iterative scaling needs on average 260 iterations and 115 ms whereas a Newton-based algorithm needs 10 iterations and 0.14 ms on a system with an Intel i7-4790 CPU. For scenarios with even more predicates iterative scaling quickly becomes too slow to be practical while our new Newton-based algorithm on the CPU and even more so on the GPU are able to calculate a solution in a few milliseconds. Hence, with our method we can avoid strategies like partitioning the set of predicates to reduce the problem size that can be found in real-world scenarios [9].

Notation	Description
$p_0, \ldots, p_{z-1}$	z predicates
$N = \{0, \ldots, z-1\}$	set of all predicate indices
$n = 2^z$	abbreviation
$T \subseteq 2^N$	set of indices of known selectivities
$m = \lvert T \rvert$	number of known selectivities
β_T	vector of known selectivities
C	complete design matrix
D	(partial) design matrix
$s(p)$	selectivity of predicate p
Bit-wise operations	Description
$\lvert$	bit-wise or
&	bit-wise and
~	bit-wise complement
$i \subseteq j$	boolean function returning $j = (i \lvert j)$

Table 1: Notation

In this paper, we propose to use a Newton-based algorithm to solve the entropy maximization problem. We formalize the problem as a series of vector- and matrix operations. However, a naive implementation of these operations fails to achieve the performance requirements for this method. Hence, we discuss in depth how to efficiently implement this algorithm and show that it is vastly superior to both iterative scaling and the naive implementation of the Newton method. We elaborate on the efficient implementation for both the CPU and the GPU.

The rest of the paper is organized as follows. Section 2 formally introduces the problem as a series of vector- and matrix operations. Section 3 describes and evaluates the (almost) straightforward implementation and the optimized implementation of Newton's algorithm for the CPU. Section 4 describes and evaluates our GPU implementation of the algorithm. Section 5 reviews how entropy maximization can be integrated into query optimizers and concludes the paper.

2 PROBLEM FORMALIZATION

In this section, we present an elegant way to formalize the maximum entropy method for selectivity estimation. This is a necessity since standard entropy maximization algorithms require a matrix-based representation of the problem, which is not yet readily available. Only this matrix-based representation of the problem will allow us to derive an efficient algorithms.

2.1 Design Matrix

Since we need a matrix representation of the problem, we need to heavily deviate from the notation of Markl et al. [9]. However, in our opinion, the resulting representation is much more elegant. From the notation of Markl et al. [9], we only keep the letter T to denote the indices of the known selectivities. For convenience, the most important parts of the notation are summarized in Table 1. The lower part contains the notation for bit-wise operations, which will be required for our efficient implementations.

2.1.1 Conjunctions of (Simple) Predicates (β). Consider a conjunctive query

$$p_0 \wedge \ldots \wedge p_{z-1}$$

of z predicates. These may be selection predicates or join predicates [9].

Let $N = \{0, \ldots, z-1\}$ be the set of numbers from 0 to $z-1$. Then, all subsets $X \subseteq N$ can be represented as a bit-vector of length z denoted by $\text{bv}(X)$ where the set bits indicate the indexes of those elements of N which are also included in the subset X. Further, this bit-vector can be interpreted as a binary number. We make no distinction between the bit-vector and the integer it represents and use whatever is more convenient. For example, we use the notation $i \subseteq j$ to denote the fact that i has a '1' only in those positions where j has a '1', i.e., $j = i|j$ holds.

For any $X \subseteq N$ define the formula

$$\beta(X) := \wedge_{i \in X}\, p_i$$

i.e., $\beta(X)$ is the conjunction of all predicates p_i whose index i is contained in X. The following table gives a complete overview for $z = 3$, where we order bits from least significant to most significant:

$\text{bv}(X)$	$\beta(X)$
1=100	p_0
2=010	p_1
3=110	$p_0 \wedge p_1$
4=001	p_2
5=101	$p_0 \wedge p_2$
6=011	$p_1 \wedge p_2$
7=111	$p_0 \wedge p_1 \wedge p_2$

where the first column gives the integer value and its bit-vector representation of the index set X and the second column the corresponding conjunction of predicates contained in X. We use $\beta(i)$ instead of $\beta(X)$ if i is the bit-vector/integer representation of some X.

The selectivity of $\beta(X)$, i.e., the probability of $\beta(X)$ being true is denoted by $\beta(X)$. A special case occurs for the empty set. The empty conjunct is always true. Thus $\beta(\emptyset) = \beta(0) = 1$.

2.1.2 Complete Conjuncts (γ). A conjunction of literals containing all predicates either positively or negatively is called *complete conjunct* (*atom* by Markl et al., also *minterm*). For $n = 3$, the following table contains a list of all complete conjuncts:

i	$\gamma(i)$				
0=000	$\neg p_0$	$\wedge$	$\neg p_1$	$\wedge$	$\neg p_2$
1=100	p_0	$\wedge$	$\neg p_1$	$\wedge$	$\neg p_2$
2=010	$\neg p_0$	$\wedge$	p_1	$\wedge$	$\neg p_2$
3=110	p_0	$\wedge$	p_1	$\wedge$	$\neg p_2$
4=001	$\neg p_0$	$\wedge$	$\neg p_1$	$\wedge$	p_2
5=101	p_0	$\wedge$	$\neg p_1$	$\wedge$	p_2
6=011	$\neg p_0$	$\wedge$	p_1	$\wedge$	p_2
7=111	p_0	$\wedge$	p_1	$\wedge$	p_2

Note that two different complete conjuncts can never be true simultaneously. The complete conjuncts have been indexed by their bit-vector representation, where a positive atom corresponds to '1' and a negative atom corresponds to '0'. For a given $X \subseteq N$, denote by $\gamma(X)$ the complete conjunct X:

$$\gamma(X) := \bigwedge_{i \in X} p_i \;\wedge\; \bigwedge_{i \notin X} \neg p_i$$

The probability of a complete conjunct $\gamma(X)$ for some X being true is denoted by $\gamma(X)$.

2.1.3 Correspondence between β and γ. For a given $X \subseteq N$, the bit-vectors y of the complete conjuncts $\gamma(Z)$ contributing to $\beta(X)$ can be expressed as all the bit-vectors y which contain a '1' at least at those positions where the bit-vector representation

bv(X) of X contains a '1'. That is

$$\{y | y \supseteq \text{bv}(X)\}.$$

Consider $X = \{0\}$ ($\hat{=}100$). Then

$$\begin{array}{rcccccccc} \beta(X) & = & s(& p_0 & \wedge & \neg p_1 & \wedge & \neg p_2 &)+ \\ & & s(& p_0 & \wedge & p_1 & \wedge & \neg p_2 &)+ \\ & & s(& p_0 & \wedge & \neg p_1 & \wedge & p_2 &)+ \\ & & s(& p_0 & \wedge & p_1 & \wedge & p_2 &), \end{array}$$

where $s(p)$ denotes the selectivity of the complete conjunct p. For $X = \{0, 1\}$ ($\hat{=}110$):

$$\beta(X) = s(p_0 \wedge p_1 \wedge \neg p_2) + s(p_0 \wedge p_1 \wedge p_2).$$

As a special case, we get for $X = \emptyset$ ($\hat{=}000$) that all complete conjuncts contribute to $\beta(\emptyset)$. Further, the sum of them must be one. Consequently, we always assume that the empty set is contained in the set of known selectivities T, i.e., $\emptyset \in T$.

2.1.4 Complete Design Matrix C. In case $T = 2^N$, all selectivities are known. Define $n = 2^z$. Then, we define the *complete design matrix* $A \in \mathbb{R}^{n,n}$ as

$$C = (c_{i,j}) = \begin{cases} 1 & \text{if } i \subseteq j \\ 0 & \text{else} \end{cases}$$

where we use indices in $[0, 2^z - 1]$. Note that C is unit upper triangular, nonsingular, positive definite, and persymmetric.

For $z = 3$, we have

$$C = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

This design matrix helps us to go from probabilities for complete conjuncts to selectivities for positive conjuncts. Let $b = (\beta(0), ..., \beta(n-1))^t$ the column vector containing all the selectivities $\beta(X)$ for all $X \in 2^N$ and $x = (\gamma(0), ..., \gamma(n-1))^t$ the column vector containing all the selectivities for all complete conjuncts. Then,

$$Cx = b$$

holds.

2.2 The (Partial) Design Matrix D

We first establish some notation to eliminate rows and columns in some matrix A. Let $A \in \mathbb{R}^{n,n}$ be some matrix. Let $T \subseteq \{0, \ldots, n-1\}$, $m := |T|$, be a set of column indices. Then, we denote by $A|_{c(T)}$ the matrix where only the columns in T are retained. Likewise, we denote by $A|_{r(T)}$ the matrix derived by retaining only the rows in T. These operations can be expressed via matrix multiplication. For an index set T with $m = |T|$, we define the matrix $E^{m,n,T} \in \mathbb{R}^{m,n}$ as

$$E_{m,n,T}(i,j) = \begin{cases} 1 & \text{if } j = T[i] \\ 0 & \text{else} \end{cases}$$

where $T[i]$ denotes the i-th element of the sorted index set T. For example, for $m = 4$, $n = 8$, $T = \{1, 3, 5, 7\}$, we get

$$E_{4,8,T} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

NewtonA($b(= \beta_T), T, \epsilon$)

```
1  w = 0
2  x = exp(−1)
3  while (δ > ε)
4      A = D diag(x) D^t
5      solve Ay = b − Dx
6      w = w + y
7      x = exp(D^t w − 1)
8      δ = ||b − Dx||
9  return (x, Cx)
```

Figure 1: Newton Variant A [2, p73]

Then, for $A \in R^{n,n}$

$$\begin{array}{rcl} A|_{r(T)} & = & E_{m,n,T} A \\ A|_{c(T)} & = & A (E_{m,n,T})^t \end{array}$$

holds. For a given subset $T \subseteq \{0, \ldots, n-1\}$ (of known selectivities), we retain only those rows from the complete design matrix C for which there is an entry in T. We define the problem specific *(partial) design matrix* D for T as

$$D := C|_{r(T)} = E_{m,n,T} C \in \mathbb{R}^{m,n} \quad (1)$$

where $m := |T|$. Clearly, the rank of D is m.

2.3 Problem Definition

For z predicates, a given vector β_T of known selectivities and indices T thereof, the idea of Markl et al. is to find the solution to $Dx = \beta_T$ that maximizes the entropy of the solution vector x [9]. That is, the problem to solve can be specified as

$$\underset{x}{\text{argmax}} \sum_{i=0}^{n-1} -x_i \log x_i \text{ subject to } Dx = \beta_T \text{ and } x \geq 0 \quad (2)$$

where $n = 2^z$. Note that, we must have that $\sum_{i=1}^{n-1} x_i = 1$, but this is implied since we assume that $\emptyset \in T$ always holds.

3 EFFICIENT CPU IMPLEMENTATION

In this section, we first discuss an implementation of Newton's algorithm to solve the entropy maximization problem that is directly derived from [2]. Due to the matrix-based formalization of our problem, the algorithm is readily applicable and we call this Variant A, and it represents the state-of-the-art implementation of Newton's algorithm. This algorithm it's rather inefficient since its steps require multiplications of large vectors and matrices. We improve this by devising a method for how these matrix and vector operations can be computed very efficiently. This leads us to Variant B of Newton's algorithm. Finally, we evaluate the runtime of both variants on an Intel CPU and compare it with the iterative scaling which was used by Markl et al.

3.1 Newton Variant A

Markl et al. propose to use iterative scaling to solve the optimization problem in Eqn. 2 [9]. However, it is well-known that iterative scaling converges very slowly [2, p82]. In contrast, a Newton-based approach exhibits local quadratic convergence [2, p73]. We thus selected a Newton-based algorithm applied to the dual problem of Eqn. 2:

$$\underset{w}{\text{argmin}} \exp(D^t w - 1)^t \vec{1} - \beta_T^t w \quad (3)$$

as the basis of our work, where we suppose that the set $\{x \in \mathbb{R}^n : Dx = \beta_T,\ x \geq 0\}$ has a nonempty interior (see also [2, p55]).

Fig. 1 shows the code of a Newton-based algorithm to solve the maximum entropy problem defined in Sec. 2.3. As input, it receives the vectors b and T of known selectivities and their indices, and some $\epsilon > 0$ used in the stop criterion. It returns the solution x maximizing the entropy and the vector Cx containing the β-selectivities for all possible conjuncts. Although T does not occur in the body of Fig. 1, it is used in the definition of the design matrix D (see Eqn. 1).

The steps in the algorithm differ vastly in complexity. The initializations of w and x have complexity $O(n)$ and $O(m)$, respectively, and are thus rather uncritical. The calculation of $w = w + z$ in Line 6 has complexity $O(m)$ and is thus rather uncritical, too.

The calculation of $A = D\text{diag}(x)D^t$ in Line 4 can be very expensive if implemented literally. Note that $\text{diag}(x)$ is a diagonal $(n \times n)$-matrix with x on its diagonal. Using standard matrix multiplication, the complexity of this step is $O(m*n^2+m^2*n)$. However $\text{diag}(x)$ contains only zero's besides the diagonal and thus a more efficient procedure which does not rely on materializing $\text{diag}(x)$ can be devised:

```
GET_DDIAGXDT(D, x)
1  for (0 ≤ i < m, 0 ≤ j < m)
2      s = 0
3      for (0 ≤ k < n)
4          s += D[i,k] * x[k] * D[j,k]
5      A(i,j) = s
6  return A
```

This procedure has complexity $O(m^2 * n)$ and is thus far better than the naive approach using matrix multiplication.

In step (5), we need to solve $Ay = b - Dx$ for y. Calculating Dx has complexity $O(m * n)$. To solve the equation, note that the (m, m) matrix $A = D\text{diag}(x)D'$ calculated in step (2) is symmetric, non-singular, and positive definite. Thus, the efficient Cholesky decomposition [5, p237] can be applied to derive a lower triangular matrix L with $A = LL^t$. Then, we derive the solution y using back substitution [4, p89]. The complexity of this procedure is $O(m^3)$.

In step (7), we need to calculate $D^t w$, which has complexity $O(m * n)$. Step (8) with complexity $O(m)$ is uncritical again, as Dx has been calculated in step (5) already.

In step (9), we need to calculate the product of the complete design matrix C with the primal solution vector x. Using standard matrix multiplication this step has complexity $O(m * n)$.

The complexities of the steps become visible when profiling Newton Variant A for $z = 8 \ldots 10$: roughly 80% of the runtime is spent in procedure GET_DDIAXDT.

3.2 Newton Variant B

As we will see in below, a careful analysis of the structure of the complete design matrix C allows us to derive a reduction-based algorithm that avoids redundant computations resulting in an algorithm for Newton's method with lower computational complexity than the state-of-the-art algorithm from Sec 3.1.

3.2.1 Recursive Characterization of C. The complete design matrix C can also be defined recursively. Denote by $C_z \in \mathbb{R}^{n \times n}$ with $n = 2^z$ the complete design matrix for z predicates. Then

$$C_0 = (1)$$

and

$$C_{z+1} = \begin{bmatrix} C_z & C_z \\ 0 & C_z \end{bmatrix}$$

characterize the complete design matrix C. Another possibility to define C is to use the Kronecker product $\otimes$ [5, p337]. With

$$C_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

we have

$$C_{z+1} = C_1 \otimes C_z$$

3.2.2 Efficient Calculation of Cx and $C^t x$. Let us turn to calculating Cx for some vector $x \in \mathbb{R}^n$, which we need to do efficiently for our Newton-based algorithm. If we cut $x \in \mathbb{R}^n$ into two halves $x_1, x_2 \in \mathbb{R}^{n/2}$, we observe that

$$C_z x = \begin{pmatrix} C_{z-1} & C_{z-1} \\ 0 & C_{z-1} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} C_{z-1}x_1 + C_{z-1}x_2 \\ C_{z-1}x_2 \end{pmatrix} \quad (4)$$

The term $C_{z-1}x_2$ occurs twice but has to be calculated only once. Based on this observation, it is easy to implement a recursive procedure calculating $C_z x$ in $O(z2^z)$, i.e. $O(n\log n)$ substituting $n = 2^z$. As a major contribution of this paper, we are now able to reduce the algorithmic complexity of the newton method from $O(n^2)$ down to $O(n\log n)$.

In order to avoid the overhead of recursion, we provide an efficient iterative algorithm. We assume that the in/out argument `Cx` has been initialized with x. Further, `vp_add` is an AVX2-based implementation to add two vectors of length h.

```
void get_Cx(double* Cx, uint z)
1  w = h = s = t = 0;
2  n = 1 << z;
3  for (w = 2; w <= n; w <<= 1) // width
4      for (s = 0; s < n; s+ = w) // start of first half
5          h = (w >> 1); // half of width
6          t = s + h; // start of second half
7          vp_add(Cx + s, Cx + t, h);
```

A procedure to efficiently calculate $C^t y$ can be devised similarly by replacing `Cx` by `Ctx` and `vp_add(Cx + s, Cx + t, h)` by `vp_add(Ctx + t, Ctx + s, h)`. We call this algorithm `get_Ctx` to w'.

3.2.3 Efficient Calculation of Dx and $D^t x$. First remember that for $n = 2^z$, z being the number of predicates, (1) the complete design matrix C is of dimension (n, n) and (2) the design matrix D is of dimension (m, n). where in typical applications m will be much smaller than $n = 2^z$.

As we have seen in Sec. 3.2.2, calculating Cx in Line 9 can be implemented very efficiently. By exploiting the definition of D in Eqn. 1, we can evaluate $Dx = E_{m,n,T}Cx$ efficiently by first calculating Cx and then picking the components contained in T. This has to be done only once to calculate the expressions Dx in Lines 5 and 8, and Cx in Line 9. Further, $C^t x$ can be calculated efficiently using algorithm `get_Ctx`. Thus, calculating $D^t w$ in step (7) can be implemented efficiently by exploiting the fact that $D^t = C^t E^T_{m,n,T}$. We can embed w into a vector w' in $\mathbb{R}^n$ via

$$w'[j] = \begin{cases} w[i] & \text{if } j = T[i] \text{ for some } i \\ 0 & \text{else} \end{cases}$$

$(0 \leq i < m,\ 0 \leq j < n)$ and apply algorithm `get_Ctx`.

3.2.4 Efficient calculation of $D\text{diag}(x)D^t$. Next, we discuss an efficient implementation of step (4). As we have already calculated Cx, we now show that it is possible to calculate $(D\text{diag}(x)D^t)$ from Cx. We start with an efficient algorithm to calculate

$$(C\text{diag}(v)C^t).$$

Observe that $(\text{diag}(v)C^t) = (C\text{diag}(v))^t$. Further,

$$(C\text{diag}(x))[j,k] = \sum_{l=0}^{n-1} c_{j,l}\text{diag}(x)[l,k] = c_{j,k}x_k$$

Thus, using

$$\begin{aligned}
(C\text{diag}(x)C^t)[i,j] &= \sum_{k=0}^{n-1} c_{i,k}(C\text{diag}(x))^t[k,j] \\
&= \sum_{k=0}^{n-1} c_{i,k}(C\text{diag}(x))[j,k] \\
&= \sum_{k=0}^{n-1} c_{i,k}c_{j,k}x_k \\
&= \sum_{(i|j)\subseteq k} x_k \\
&= (Cx)[i|j]
\end{aligned}$$

we can calculate $(C\text{diag}(x)C^t)$ from Cx. Since

$$\begin{aligned}
D\text{diag}(x)D^t &= (E_{m,n,T}C)\text{diag}(x)(E_{m,n,T}C)^t \\
&= E_{m,n,T}(C\text{diag}(x)C^t)E^t_{m,n,T}
\end{aligned}$$

we can use Cx to fill $(D\text{diag}(x)D^t) \in \mathbb{R}^{m,m}$ via

$$(D\text{diag}(x)D^t)[i,j] = (Cx)[T[i] \mid T[j]] \tag{5}$$

for $0 \le i,j < m$.

3.3 Evaluation

In order to evaluate the implementations of the two variants of Newton's algorithm and the iterative scaling used by Markl et al., we need to generate entropy maximization problems. Since generation of β selectivities easily leads to inconsistencies, we generate a random vector x of size n containing positive integers interpreted as cardinalities for all complete conjuncts γ. Dividing each x_i by $\sum_i x_i$ results in γ-selectivities. Calculating $b = Cx$ results in a complete set of consistent β-selectivities. From these, we select the subset T of known selectivities by extracting selectivities for single predicates and conjunctions of two or three predicates. In practice, not all pairs or triples will be available. Thus, the runtimes reported in the experiments below can be seen as loose upper bounds on the runtime in practice.

We use the stopping criterion $||b/Dx||_q \le \epsilon$ where b/Dx denotes component-wise division,

$$||y||_q := \max_i(\max(y_i, 1/y_i)),$$

and $\epsilon = 1 + 10^{-8}$.

We implemented iterative scaling and the two variants of our Newton-based algorithm in C++ and compiled them with g++ version 7.2.1 with option -O3. The experiments where run on a system with an Intel i7-4790 CPU. Note that this CPU with Haswell architecture had a better single-thread performance than a newer server CPU with Skylake architecture. We report the average execution time of 777 generated problems for each number z of predicates. Our implementation runs in single-threaded mode.

Figures 2 and 3 show the average runtime of our CPU implementation versus the runtime of iterative scaling (as proposed by Markl et al. [9]) if the set of known selectivities T contains all unary and additionally all binary or ternary conjuncts. Column z contains the number of predicates considered and column m contains the number of known selectivities. Besides the average runtime in milliseconds, we include the average number of iterations.

As one can see, our Newton-based implementation is much more efficient than the originally proposed iterative scaling algorithm. For ten conjuncts the runtime of iterative scaling already exceeds one second. Further, as expected, Newton Variant B is much more efficient than Newton Variant A.

		Newton Var. A	Newton Var. B		Iterative Scaling	
z	m	runtime [ms]		#itr	runtime [ms]	#itr
3	7	0.009	0.004	7.3	0.14	190
4	11	0.017	0.008	7.8	0.47	190
5	16	0.061	0.027	8.1	2.1	200
6	22	0.23	0.048	9	9.4	210
7	29	0.84	0.075	9.1	34	240
8	37	2.9	0.14	10	120	260
9	46	10	0.25	11	370	280
10	56	29	0.41	11	1100	310
11	67	98	0.73	12	—	—
12	79	310	1.4	13	—	—
13	92	1000	2.7	13	—	—
14	106	3300	5.3	14	—	—
15	121	11000	11	15	—	—
16	137	—	23	15	—	—
17	154	—	48	16	—	—
18	172	—	100	17	—	—
19	191	—	200	17	—	—
20	211	—	480	18	—	—

(Intel i7-4790, single-threaded, $T = \{t|\text{popcnt}(t) \le 2\}$)

Figure 2: Newton vs. Iterative Scaling

		Newton Var. A	Newton Var. B		Iterative Scaling	
z	m	runtime [ms]		#itr	runtime [ms]	#itr
4	15	0.04	0.02	8.7	3	890
5	26	0.15	0.05	9	16	910
6	42	0.79	0.13	9.3	79	1000
7	64	3.9	0.33	10	360	1200
8	93	16	0.76	10	1600	1400
9	130	65	1.8	11	6700	1580
10	176	230	4.1	11	26000	1800
11	232	890	9.1	12	—	—
12	299	3400	20	13	—	—
13	378	11000	40	13	—	—
14	470	38000	80	14	—	—
15	576	120000	150	15	—	—
16	697	—	270	15	—	—
17	834	—	480	16	—	—
18	988	—	880	17	—	—
19	1160	—	1400	17	—	—
20	1351	—	2600	18	—	—

(Intel i7-4790, single-threaded, $T = \{t|\text{popcnt}(t) \le 3\}$)

Figure 3: Newton vs. Iterative Scaling

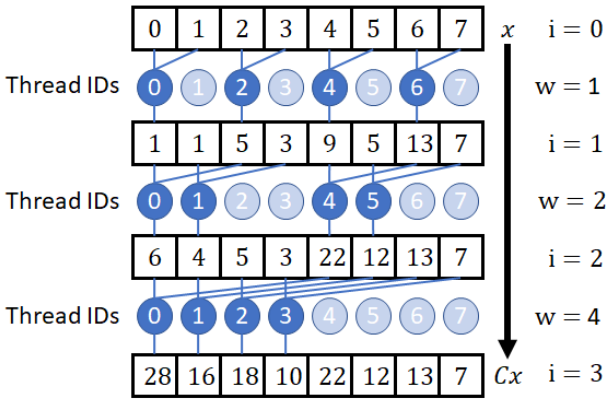

Figure 4: Scheme of the efficient GPU implementation of Cx for an initial $x = 0, 1, \ldots, 7$

Figure 2 shows that for up to ten predicates, the runtime to calculate all selectivities needed by the query optimizer is below 0.5 milliseconds if Newton Variant B is used, i.e. three orders of magnitude faster than iterative scaling. However, somewhere between 11 and 20 predicates, depending on the context (e.g., ad hoc queries vs. repeated execution), even the runtimes of our optimized Newton Variant B implementation becomes too high. In particular, Figure 2 indicates that the Newton Variant B with 13 predicates exceeds one second of runtime, while our new method B finishes even 20 conjuncts in less than a second.

In general the runtimes are higher when the problems contain all unary, binary or ternary conjuncts. As can be seen in Figure 3, iterative scaling needs almost 26 seconds for 10 predicates while our new Newton method with Variant B calculates these problems in about 4 milliseconds, i.e. more than 4 orders of magnitude faster. The naive implementation of Newton's method in Variant A needs more then ten seconds runtime for 15 predicates while Variant B is nearly 4 orders of magnitude faster for the same problems.

The increasing runtimes as we consider more and more complex predicates motivated us to pursue a GPU implementation.

4 EFFICIENT GPU IMPLEMENTATION

In this section we describe how the Newton algorithm can be implemented efficiently on a modern GPU. We first explain the multi-threaded GPU implementation of Variant B presented in Sec. 3.2. After that we present experimental results of our implementation using CUDA 10.0 on an NVIDIA Tesla V100 GPU.

4.1 Newton Variant B on the GPU

We discuss how Variant B of Newton's method can be implemented on an NVIDIA GPU. We focus our presentation on the implementation of Cx because, as we have seen in Sec. 3.2, this operation is at the heart of the implementation of steps (4), (7), (8) and (9) of the Newton algorithm presented in Figure 1. We also point out how the remaining step (5), the Cholesky decomposition, is implemented efficiently on the GPU. Finally we outline how we organize our code in kernels of the end-to-end implementation.

4.1.1 GPU Implementation of Cx and $C^t x$. As the NVIDIA V100 GPU used in our experiments offers an abundance of 5120 CUDA cores, we need to extend the implementation of `get_Cx` presented in Sec. 3.2.2 to support massive multi-threading. Figure 4 illustrates the parallelization scheme we use in our implementation. Here, the required operations for calculating Cx are shown for $x = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and $z = 3$. Boxes represent the contents of x after each iteration i, and dark (light) blue circles represent active (inactive) CUDA threads. In each iteration, every active thread performs one addition and stores the result. The connecting blue lines indicate the flow of data. In every iteration half of the GPU threads are active while the other half is idle. While this may seem wasteful, it allows us to use a simple mapping from thread-id to accessed memory addresses. A more effective use of the GPU threads would require a more complex mapping. In fact, we did not find an efficient way to map thread-ids to memory addresses while keeping all threads active all the time. As the maximum number of threads per thread block for the Tesla V100 is 1024, the first ten iterations of our scheme can be performed without requiring communication between different thread blocks. During these ten iterations we make use of the GPUs shared memory, and access to global memory is only required once when loading x into shared memory and once when writing Cx back to global memory. This is beneficial because compared to global memory, shared memory on the NVIDIA V100 GPU offers lower latency and significantly higher bandwidth. Hence, for $z \leq 10$ we use the kernel using shared memory shown in Listing 1. In every iteration of the outer loop we advance with processing vector x by the number of available threads. Note, that for z predicates we have $n = 2^z$ elements to process, i.e. for $z = 15$ we have $2^{15} = 32768$ elements to process. The inner loop in Listing 1 adds the elements as illustrated in Figure 4.

For $z > 10$, no efficient shared memory implementation is possible as threads of one thread block would need to access shared memory allocated in another thread block. This is not possible, and as a consequence all memory accesses have to go to global memory. This requires global synchronization through individual kernel launches. We call this global kernel to compute Cx once for every $z > 10$. It is shown in Listing 2. In our implementation we use templates to generate these calls at compile time. The parameter `direction` allows us to not only calculate Cx but also to calculate $C^t x$. When `direction` is set to 1, the algorithm proceeds backwards, giving us $C^t x$. This is needed in step (7) of the Newton algorithm where we use the product $D^t w$.

Recall that steps (4), (7), (8) and (9) in the Newton algorithm shown in Figure 1 build upon or use the calculation of Cx. This is why we do not describe the implementation of these steps in detail here. The basic ideas are similar to the ones presented for the computation of Cx.

4.1.2 Cholesky Solver. As for the CPU implementation presented in Sec 3.1, solving $Ay = b - Dx$ for y in step (5) of the Newton algorithm shown in Figure 1 can be done using Cholesky decomposition [5, p237]. Fortunately, we can use the cuSolver library from the CUDA toolkit [16] for large problems, i.e. for $m \geq 40$. First, we rely on `cusolverDnDpotrf` to factorize A in a kernel call. Then, we call the kernel `cusolverDnDpotrs` where we pass $b - Dx$ as argument and get y as result of step (5).

As multiple kernel calls are involved in these steps, and each kernel call implies a call overhead of approximately $5 - 10\mu s$, we also implement a variant of the Cholesky decomposition using only a single kernel call. We use this kernel as a solver for small problems, i.e. $m < 40$. The implementation is based on [13] and calculates the solution of the system of equations via

Listing 1: Kernel to compute Cx in shared memory

```
template <int BLOCK_SIZE_X>
__global__ void getCxShared(double* __restrict__ const x, const unsigned int z,
                            const bool direction=0) {
    unsigned int stride =  blockDim.x * gridDim.x;
    __shared__ double xShared[BLOCK_SIZE_X];
    unsigned int end = (1<<z);
    for(int globalIdx = threadIdx.x+blockDim.x*blockIdx.x; globalIdx < end; globalIdx+=stride) {
        xShared[threadIdx.x] = x[globalIdx];
        __syncthreads();
        for(int w = 1; w < BLOCK_SIZE_X; w<<=1) {
            if((threadIdx.x/w)%2 == direction) {
                xShared[threadIdx.x]+=xShared[threadIdx.x+w-direction*2*w];
            }
            __syncthreads();
        }
        x[globalIdx] = xShared[threadIdx.x];
    }
}
```

Listing 2: Kernel to compute Cx in global memory

```
template <unsigned int iteration>
__global__ void getCxGlobal(double* __restrict__ const x, const bool direction=0) {
    static constexpr auto offset = 1U << iteration;
    const int myGlobalIdx = threadIdx.x+blockDim.x*blockIdx.x;
    const int blockOffset = (blockIdx.x*1024/offset)*2*offset;
    const int myElementIdx = offset*direction + blockOffset + myGlobalIdx%offset;
    x[myElementIdx]+=x[myElementIdx+offset-2*direction*offset];
}
```

Gaussian elimination without pivoting. It is implemented to run in a single thread block using shared memory. In our experiments, this reduced the end-to-end runtimes of the Newton algorithm by $0.2 - 0.4ms$. However, as the CPU implementation is still faster than the GPU for such small problems this alternative is not really needed.

4.1.3 End-To-End GPU Implementation. We now describe how the various kernels are combined to implement Newton's algorithm on the GPU. In Figure 5, we can only present pseudo code as all the GPU code taken together is several hundred lines long. The initialization in steps (1) - (3) and the main loop are realized in function `NewtonB_GPU`.

While the logic of the loop is the same as in Figure 1 for the CPU code we organize the code to minimize the number of kernel calls. For example, in step (5) we compute both $D\text{diag}(x)D^t$ and also $b - Dx$ in a single kernel call to `buildMatrixA`. In this kernel we first compute Cx calling `getCxShared` and then, if $z > 10$, we call `getCxGlobal` in a loop for every $10 < w \leq z$. In the second step of kernel `buildMatrixA`, we gather from Cx the elements for Dx and $A = D\text{diag}(x)D^t$ as explained in Sec 3.2.3 and Eqn 5 in Sec 3.2.4. In Sec 4.1.2 we explain how we implement step (6) of the loop in function `NewtonB_GPU`, i.e. using the cuSolver library of CUDA for larger problems. Step (7) computes $w = w - y$ using thrust::transform from Thrust, the CUDA C++ template library [16]. Then, step (8) fuses steps (7) and the computation of $b - Dx$ in step (8) of the CPU-based code from Figure 1 into a single kernel `productOfDtw`. This kernel first distributes vector w into x, and then `productOfDtw` uses the logic of `get_Cx_GPU` to compute $D^t w$ using direction = 1 as parameter to handle the transposed matrix; see Sec 3.2.3. As part of this computation we can also calculate the vectors u_{old}, u_{new} and x in the same kernel. Notice, that after the call to `productOfDtw` the vector u_{old} contains the element-wise delta of the last loop iteration. We use this vector in step (9) to determine δ to check for convergence of the algorithm. In our GPU implementation we use the L_∞ norm and $\epsilon = 10^{-8}$. Because of the local quadratic convergence of the Newton algorithm we found that the norm used to check for convergence had virtually no impact on the convergence of the algorithm. If convergence is reached, we return the solution of the Newton algorithm in step (11) by doing one final call to get_Cx_GPU(x,0).

4.2 Evaluation

To evaluate the performance of the GPU-based implementation of the Newton algorithm presented in Sec 4.1, we generated the same entropy maximization problems as in Sec 3.3. We compiled the Newton algorithm using gcc 7.3.1 for the host code and CUDA 10.0 for the kernels on the GPU and compiled them with g++ -O3

The experiments where run on a system with an Intel Xeon E7-8890v3, i.e., using a CPU from the same hardware generation as we used for the evaluation of the CPU-based implementation in Sec 3.3. The system was equipped with a PCI-attached NVIDIA Tesla V100 GPU with 16GB of HBM2 memory. We report the average execution time of the generated problems for different numbers of predicates, z. During the experiments, the host code on the CPU was running in a single thread; virtually all computation was done on the GPU. We remark that the runtimes for the CPU implementations reported in Sec 3.3 used a single thread on the host. The GPU implementation we used here performs busy waiting on the host. With CUDA 10.1 the graph feature

```
GET_Cx_GPU(x, direction)
1  y = getCxShared(x,direction)
2  for w = 1 to z − 10
3      Cx = getCxGlobal<10 + w>(y,direction)
4  return Cx

BUILDMATRIXA(b, x)
1  Cx = get_Cx_GPU(x,0)
2  (A, Dx) = distribute Cx to A and Dx using Sec 3.2.3 and Sec 3.2.4
3  return (A, Dx)

PRODUCTOFDTW(w)
1  D^t w = 0
2  distribute w into x
3  D^t w = get_Cx_GPU(x,1)
4  together with get_Cx_GPU(x,1), in the same kernel also compute
5      x = exp(−D^t w)
6      u_new = x/exp(1)
7      u_old = u_old − u_new
8  return (D^t w, u_old, u_new, x)

NEWTONB_GPU(b(= β_T), T, ϵ)
1  w = 0
2  b = b ∗ exp(1)
3  x = 1
4  while (δ > ϵ)
5      (A, Dx) = buildMatrixA(b, x)
6      solve Ay = b − Dx for y using cuSolver
7      w = w − y
8      (D^t w, u_old, u_new, x) = productOfDtw(w)
9      δ = ||u_old||∞
10     swap(u_old, u_new)
11 return (get_Cx_GPU(x,0))
```

Figure 5: GPU version of Newton Variant B

became available which allows to model the graph of kernels and reduce the call overheads for the kernels. Furthermore, the graph recapture feature introduce with CUDA 10.2. supports passing parameters to these graphs further reducing the call overheads of the GPU. With this the GPU implementation may become faster for smaller problems, but an initial overhead to create and instantiate the graph of about $0.4ms$ would remain. For larger problems these overheads become insignificant.

In Figure 6 we present the runtime for configurations with different complexity. As in Sec 3.3 column z contains the number of predicates considered and column m contains the number of known selectivities.

The runtimes in the third column in Figure 6 are reported for problems where the set of known selectivities T contains all unary and additionally all binary conjuncts. In this setup, the GPU is faster than the fastest CPU implementation for 13 or more predicates. For 20 predicates the runtime of the fastest CPU implementation was 480 ms (see Figure 2) while the GPU implementation only needs 18 ms, i.e. speed-up of 27x. Furthermore, the NVIDIA V100 GPU is able to compute problems with 25 predicates in only 632 ms. In comparison, the state-of-the art method based on iterative scaling presented by Markl et al. [9] already needs more than one second to compute the result for only 10 predicates (see Figure 2).

The runtimes in the fifth column in Figure 6 refer to problems where the set of known selectivities T contains all unary, binary

	Newton GPU			
z	m	runtime [ms]	m	runtime [ms]
3	7	0.9	8	1.0
4	11	0.7	15	0.9
5	16	0.7	26	0.9
6	22	0.7	42	1.2
7	29	0.8	64	1.4
8	37	1.0	93	1.8
9	46	1.3	130	2.9
10	56	1.5	176	3.5
11	67	1.8	232	4.9
12	79	2.2	299	6.5
13	92	2.5	378	8.8
14	106	3.1	470	11
15	121	3.7	576	16
16	137	4.7	697	20
17	154	6.2	834	28
18	172	7.7	988	33
19	191	11	1160	46
20	211	18	1351	63
21	232	35	1562	90
22	254	63	1794	130
23	277	140	2048	220
24	301	310	2325	420
25	326	630	2626	760

(NVIDIA Tesla V100)

Figure 6: GPU implementation of Variant B of Newton's algorithm

and ternary conjuncts. Here, the GPU is faster than our fastest CPU-based implementation for 10 or more predicates. For 20 predicates the GPU-based implementation is more than 43 times faster than our fastest CPU-based implementation. Such a complex problem could not be solved in a reasonable time by the state-of-the art method based on iterative scaling [9]. According to Figure 3, that implementation processed problems with 10 predicates in almost 26 seconds while our GPU-based implementation finishes this task in only 3.5 ms, i.e. almost five orders of magnitude faster.

5 DISCUSSION AND CONCLUSION

Query optimizers rely on several sources to estimate the selectivity of complex conjunctive predicates. Many database systems use elaborate methods to serve selectivity estimation, e.g., multi-column histograms [12], samples [3], statistics on views [7] or even query feedback [14].

Entropy maximization as proposed by Markl et al. [9] considers all available information to derive a consistent estimate for all partial conjuncts of a predicate. However, as the runtimes for iterative scaling are prohibitively high already for 8 predicates, Markl et al. suggests to partition the problem into smaller conjuncts assuming independence between the selectivities of the predicates of the partitions. This risks loosing valuable information from the set of known selectivities.

With the formalization of the entropy maximization problem as a series of vector- and matrix operations we are able to derive efficient implementations for this problem using the Newton algorithm. As our CPU based algorithm is more than 4 orders of magnitude faster than the iterative scaling for the most complex problem it could handle, entropy maximization becomes a feasible

option even for complex predicates without sacrificing the quality of the cardinality estimates. Even more, the new implementations can be applied to conjuncts with 18 predicates for the CPU or even 25 predicates for the GPU with runtimes of less than a second making partitioning the input problem irrelevant for virtually all scenarios. While Markl et al. already explained in detail how to integrate the maximum entropy method into query optimizers, we conclude that using the implementation techniques presented in this paper, entropy maximization is ready to be included into production-grade database management systems.

REFERENCES

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

[2] S.-C. Fang, J. R. Rajasekera, and H. S. J. Tsao. 1997. *Entropy Maximization and Mathematical Programming.* Kluwer.

[3] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. 2007. Maintaining bernoulli samples over evolving multisets. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS).* 93–102.

[4] G. Golub and C. van Loan. 1996. *Matrix Computations.* The John Hopkins University Press. Third Edition.

[5] D. Harville. 2008. *Matrix Algebra from a Statistician's Perspective.* Springer.

[6] Y. Ioannidis. 2003. The History of Histograms (abridged). In *Proc. Int. Conf. on Very Large Data Bases (VLDB).* 19–30.

[7] P.-Å. Larson, W. Lehner, J. Zhou, and P. Zabback. 2007. Cardinality estimation using sample views with quality assurance. In *Proc. of the ACM SIGMOD Conf. on Management of Data.* 175–186.

[8] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[9] V. Markl, P.J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T.M. Tran. 2007. Consistent Selectivity Estimation Via Maximum Entropy. *VLDB Journal* 16, 1 (January 2007), 55–76.

[10] G. Moerkotte, M. Montag, A. Repetti, and G. Steidl. 2015. Proximal operator of quotient functions with application to a feasibility problem in query optimization. *J. Computational Applied Mathematics* 285 (2015), 243–255.

[11] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. *PVLDB* 11, 9 (May 2018), 1016–1028. https://doi.org/10.14778/3213880.3213882

[12] V. Poosala and Y. Ioannidis. 1997. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proc. Int. Conf. on Very Large Data Bases (VLDB).* 486–495.

[13] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. 1988. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA.

[14] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 19–28.

[15] Xiaohui Yu, Nick Koudas, and Calisto Zuzarte. 2006. HASE: a hybrid approach to selectivity estimation for conjunctive predicates. In *International Conference on Extending Database Technology.* Springer, 460–477.

[16] CUDA Developer Zone. 2019. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/.