Computationally Reproducible

# Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation

Max Heimel
Technische Universität Berlin
max.heimel@tu-berlin.de

Martin Kiefer
Technische Universität Berlin
kiefer@campus.tu-berlin.de

Volker Markl
Technische Universität Berlin
volker.markl@tu-berlin.de

## ABSTRACT

Quickly and accurately estimating the selectivity of multidimensional predicates is a vital part of a modern relational query optimizer. The state-of-the art in this field are multidimensional histograms, which offer good estimation quality but are complex to construct and hard to maintain. Kernel Density Estimation (KDE) is an interesting alternative that does not suffer from these problems. However, existing KDE-based selectivity estimators can hardly compete with the estimation quality of state-of-the art methods.

In this paper, we substantially expand the state-of-the-art in KDE-based selectivity estimation by improving along three dimensions: First, we demonstrate how to numerically optimize a KDE model, leading to substantially improved estimates. Second, we develop methods to continuously adapt the estimator to changes in both the database and the query workload. Finally, we show how to drastically improve the performance by pushing computations onto a GPU.

We provide an implementation of our estimator and experimentally evaluate it on a variety of datasets and workloads, demonstrating that it efficiently scales up to very large model sizes, adapts itself to database changes, and typically outperforms the estimation quality of both existing Kernel Density Estimators as well as state-of-the-art multidimensional histograms.

## 1. INTRODUCTION

Estimating the cardinality of intermediate result sets is an integral part of the query optimization pipeline in a modern database: The optimizer uses these estimates to make assumptions about the costs of candidate plans [21]. Since the estimation quality directly impacts plan quality [35], incorrect estimates can cause unexpectedly bad query performance [10, 29]. An important subproblem of cardinality estimation is to compute the selectivity of multidimensional range queries over real-valued attributes [14]. Multiple authors have suggested special estimators for this case – usually variants of multidimensional histograms [7, 14, 34].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*SIGMOD'15,* May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2749438.

While multidimensional histograms generate very good estimates, they also have shortcomings that have prevented major commercial adoption. In particular, they are often complex to construct, and hard – or even impossible – to maintain under updates. An alternative class of estimators are *Kernel Density Estimators* (KDEs), which compute selectivities from a random data sample by averaging local probability distributions – the kernels – that are centered on the sampled items. A KDE model is very simple: Essentially it just consists of a data sample, making model construction and maintenance extremely easy. Since it also has statistical advantages [1], multiple authors suggested using KDE to estimate range selectivities [4, 14].

However, existing KDE-based selectivity estimators are still immature and can hardly compete with state-of-the-art multidimensional histograms. There are three major reasons for this: First, they are expensive to evaluate, given that computing an estimate requires to scan the complete sample. Second, their estimation quality is often suboptimal since they do not adjust the *bandwidth* parameter, which has a strong impact on estimation quality [1, 4, 14]. Finally, to the best of our knowledge, the question of how to maintain a KDE-based selectivity estimator under database changes has not been covered by the literature so far.

In a prior publication, we already discussed how to tackle the first problem by offloading computations to a graphics card [16]: KDE can be efficiently parallelized, and the required computations are well-suited to be accelerated by a GPU. In this paper, we substantially extend upon their work and approach the other two problems: Selecting the optimal bandwidth parameters and maintaining the model under database updates and workload changes. In particular, we make the following contributions:

1. We demonstrate how to drastically improve the estimation quality of a KDE-based selectivity estimator by solving an optimization problem over query feedback to pick the optimal bandwidth. We also show how to automatically adjust the bandwidth to workload changes via online learning.

2. We introduce a method to efficiently maintain the underlying data sample under database updates by tracking the impact of individual sample points on estimation quality and selectively replacing "bad" points.

3. We provide a GPU-accelerated implementation[1] of our estimator integrated into Postgres and evaluate it on a variety of synthetic and real-world datasets.

[1]The source code is available at: `goo.gl/aQSQNd` .

The remainder of this paper is structured as follows: In the next section, we discuss related work and introduce required concepts. The subsequent two sections provide a theoretical overview of our self-tuning, optimal KDE-based selectivity estimator, with Section 3 introducing the idea of numerically optimizing the bandwidth, and Section 4 providing details on how to automatically adjust the model. In Section 5, we give an overview of our implementation and discuss details of the GPU-acceleration. Finally, Section 6 presents the experimental evaluation, Section 7 summarizes our findings, and Section 8 concludes the paper by discussing possible directions for future work.

# 2. BACKGROUND & RELATED WORK

## 2.1 Problem Description

Given a relation $R$ with attributes $(A_1, \ldots, A_d)$, and an arbitrary query region $\Omega \subseteq D_1 \times \ldots \times D_d$ – where $D_i$ denotes the domain of attribute $A_i$ –, selectivity estimators aim to estimate the fraction $|\sigma_{\vec{x} \in \Omega}(R)|/|R|$ of tuples from $R$ that fall within $\Omega$. In this paper, we discuss the subproblem where $\Omega$ is a hyper-rectangle, i.e., it is the Cartesian product of intervals within the d attribute domains: $\Omega = (l_1, u_1) \times \ldots \times (l_d, u_d)$. Furthermore, we assume that all attributes are real-valued, meaning that $\Omega \subseteq \mathbb{R}^d$. Note that – contrary to the authors of [14] –, we do not make any further assumptions about the attribute domains. In particular, we do not assume that the data in $R$ is chosen from a known region, such as the unit cube.

## 2.2 Multidimensional Selectivity Estimators

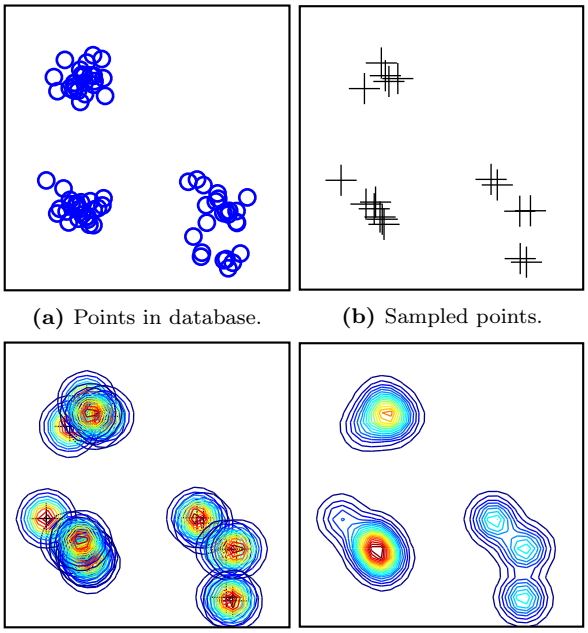
The easiest way to estimate the selectivity of a multidimensional range query is to assume that attributes are independent of each other. In this case, a $d$-dimensional estimate can be computed by multiplying $d$ one-dimensional estimates, e.g. obtained from histograms. However, since real datasets are almost always correlated, this *attribute-value independence assumption* often leads to significant estimation errors in real-world cases. These errors can cause incorrect decisions by the query optimizer, which in turn result in bad query performance [10].

Improving the quality of multidimensional selectivity estimates is a classical database research problem, and authors have actively investigated and suggested methods for over 15 years. Most suggested methods rely on multidimensional histograms, which partition the data space into buckets and track the number of tuples within those [32]. Prominent representatives include *MHIST* [34], *Genhist* [14], *STHoles* [7], and ISOMER [38]. Other discussed methods include wavelets [30], discrete cosinus transformations [26], kernel methods [4] and sampling [25, 28].

For a much more detailed overview, we refer the reader to the following two surveys: In [20], the authors give an historic overview over the development of histograms and other estimator techniques. In [14], the authors provide an experimental comparison of the selectivity estimation quality of kernel methods, wavelets, sampling techniques, and multidimensional histograms on real-valued attributes.

## 2.3 Kernel Density Estimation

One approach for multidimensional selectivity estimation are multivariate *Kernel Density Estimators*. Kernel Density Estimation is a well-known and appreciated tool from statistics that has been around since the late 1950s [1]. It is a data-driven technique to estimate a probability distribution from a data sample [1]. Figure 1 visualizes the method for a two dimensional dataset: Given a data sample (Figure 1(b)), local probability distributions – known as *kernels* – are centered around the sample points (Figure 1(c)). The estimate is then computed by averaging these local contributions (Figure 1(d)). Essentially, this amounts to sampled points contributing *probability mass* to their immediate neighborhood, which is why KDE assigns high probability to regions that lie in the vicinity of sampled data points. Kernel Methods are generally accepted to be among the most accurate estimators in statistics literature [1], and – compared to alternative methods – they offer several advantages:



**(a)** Points in database. **(b)** Sampled points.

**(c)** Kernels. **(d)** Estimated distribution.

**Figure 1:** A Kernel Density Estimator approximates the underlying distribution of a given dataset (a) by picking a random sample of data points (b), centering local probability distributions (kernels) around the sampled points (c), and averaging those local distributions (d).

- KDE has been shown to converge faster to the underlying distribution than histograms do [1]. Furthermore, compared to methods that "naïvely" evaluate the query on a sample [25, 28], KDE has been shown to consistently offer superior estimation quality [14].

- KDE models are easy to construct and to use: After the sample is collected, they are ready to be used. In particular, they do not require any additional model assumptions, like splitting or bucketization rules.

- Since a KDE model is inherently a data sample, the estimator implicitly follows any data distribution without having to explicitly model the domain space. This makes KDE very robust against effects from correlated or degenerate data.

- Compared to histograms, maintaining KDE-based selectivity estimators under database updates is comparably easy: KDE models consist primarily of a data sample, which is why maintaining them is identical to the well-understood sample maintenance problem [13].

Despite these advantages, the amount of database research literature published on KDE-based selectivity estimation is surprisingly scarce. The first paper that suggested KDE to predict selectivities – albeit only for the single-dimensional case – was published in 1999 by Blohsfeld et. al. [4]. They experimentally demonstrated that KDE is generally preferable over a purely sample-based estimator, matches the estimation quality of equi-width histograms and can drastically outperform them in case of sufficiently smooth data. In 2005, Gunopulos et. al. generalized KDE-based selectivity estimation for the multidimensional case and compared it against Genhist, their variant of a multidimensional histogram [14]. They demonstrated that KDE offers comparable estimation quality to histograms in the multidimensional case and also highlighted the comparably cheap construction costs. While those are the only publications that primarily discuss aspects of KDE-based selectivity estimation, there are publications that suggest further use-cases for KDE, including approximate range query processing [15], estimating stream cardinalities [18], online outlier detection [39], and predicting the results of skyline queries [46].

## 2.4 GPU-assisted Selectivity Estimation

A major disadvantage of KDE is its performance impact: For each estimate, the whole data sample has to be scanned, which can be quite expensive. Since database systems need to trade off between the time spent on query optimization and execution, the improved estimation quality offered by KDE is often simply not worth the additional cost.

In a prior publication, we suggested a possible approach to solve this problem by using a graphics card as a *statistical co-processor* that computes selectivity estimates [16]. Modern graphics cards bundle high-bandwidth memory with a massively parallel processor, making them an interesting platform to accelerate data-intensive operations. In particular, they are well-suited to accelerate the types of computations encountered in Kernel Density Estimation[2], and we could observe roughly a ten-fold speedup compared to a CPU implementation when running KDE on an off-the-shelf graphics card.

Besides being faster, pushing selectivity estimation to a graphics card also has further advantages: First, it allows the database to perform other tasks while selectivities are computed. Second, it enables us to use much larger model sizes within the same time budget. Since the quality of an estimator is directly proportional to its model size [1], this directly results in better estimates. Interestingly, since better estimates usually improve plan quality [21], running selectivity estimation on a graphics card is one of the only applications of GPUs in database systems that can also accelerate traditional disk-based systems [16].

[2]Note that increased transfer across the PCI Express bus – which is typically the bottleneck of any GPU-accelerated application – is of little concern: The sample is kept on the graphics card at all times, meaning that the only required transfers are the query bounds and the computed estimate.

# 3. OPTIMAL KDE MODELS

In this section, we introduce how to build optimal KDE-based selectivity estimators by adjusting model parameters over query feedback.

## 3.1 The Formal View

Formally, based on a sample $\mathcal{S} = \left\{\vec{t}^{(1)}, \ldots, \vec{t}^{(s)}\right\}$ of size $s$ from a $d$-dimensional dataset, multivariate KDE defines an estimator $\hat{p}_H(\vec{x}) : \mathbb{R}^d \to \mathbb{R}$ that assigns a probability density to each point $\vec{x} \in \mathbb{R}^d$. The estimator is defined as:

$$\hat{p}_H(\vec{x}) = \frac{1}{s \cdot |H|} \sum_{i=1}^{s} K\left(H^{-1}\left[\vec{t}^{(i)} - \vec{x}\right]\right) \tag{1}$$

In this equation, $K : \mathbb{R}^d \to \mathbb{R}$ denotes the *kernel function*, which defines the shape of the local probability distributions. The matrix $H \in \mathbb{R}^{d \times d}$ is the *bandwidth matrix*, which controls the spread of the local probability distributions.

### 3.1.1 KDE-based Range Selectivity Estimation

The probability density $\hat{p}_H(\vec{x})$ from equation (1) can be interpreted as the likelihood of finding a tuple at point $\vec{x}$. In order to approximate the selectivity $\hat{p}_H(\Omega)$ for a query region $\Omega$, we have to integrate (1) over the region:

$$\begin{aligned}\hat{p}_H(\Omega) &= \int_\Omega \hat{p}(\vec{x})\, d\vec{x} \\ &= \frac{1}{s}\sum_{i=1}^{s} \underbrace{\int_\Omega \frac{K\left(H^{-1}\left[\vec{t}^{(i)} - \vec{x}\right]\right)}{|H|} d\vec{x}}_{=\hat{p}_H^{(i)}(\Omega)}\end{aligned} \tag{2}$$

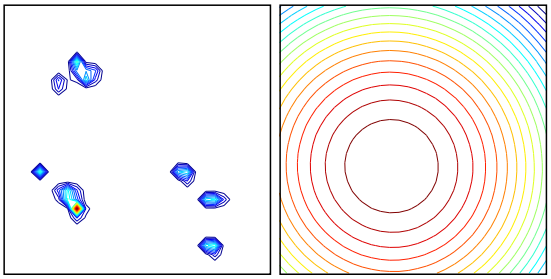
In this equation, $\hat{p}_H^{(i)}(\Omega)$ denotes the *individual probability mass contribution* of the $i$-th sample point to region $\Omega$. Equation (13) – derived in Appendix Section B – provides the closed-form expression. The selectivity estimate is then simply the average contribution across all sample points.

### 3.1.2 The Kernel Function

The *kernel function* $K : \mathbb{R}^d \to \mathbb{R}$ in equation (1) defines the shape of the local probability distributions – any function that defines a symmetric probability distribution is a valid choice. Since the shape of the kernel function barely impacts estimation quality [1], $K$ is often chosen based on certain desired properties. In our case, we require that the Kernel is continuously differentiable: Possible choices include the *Gaussian* – a standard normal distribution –, or the *Epanechnikov*, a truncated second-order polynomial.

### 3.1.3 The Bandwidth

Selecting a good *bandwidth matrix* $H \in \mathbb{R}^{d \times d}$ is much more important than choosing the kernel function [1, 23]. The bandwidth controls the spread of the local distributions, i.e., the larger its magnitude is, the further will sample points distribute probability mass into their neighborhood. Figure 2 illustrates the effect of choosing different bandwidth parameters for the estimator from Figure 1: If the bandwidth is chosen too small (Figure 2(a)), the estimator isn't smooth enough, resulting in a very spiky distribution that overfits the sample. On the other hand, if the bandwidth is chosen too large (Figure 2(b)), the estimator is smoothed too strongly, losing much of the local information and underfitting the distribution.



**(a)** Bandwidth too small. **(b)** Bandwidth too large.

**Figure 2:** Choosing the bandwidth has a crucial impact on estimation quality. If the bandwidth is too small (a), the estimator overfits the sample. If it is too large (b), all local information is lost.

A typical simplification – which we also adopt in this paper – is to assume that the bandwidth matrix $H$ is diagonal, i.e. $H = diag(h_1, \ldots, h_d)$, where $h_i$ is a scalar that controls the spread of the local distributions along the $i$-th attribute. While this simplification can have an impact on estimation quality [44], it vastly simplifies operations and allows us to derive closed-form expressions that do not exist for full bandwidth matrices [1].

## 3.2 The Bandwidth Selection Problem

Selecting the optimal – or even just a "good enough" – bandwidth is a challenging problem that has been under heavy investigation by the statistics community [1, 23]. The primary goal behind this *bandwidth selection problem* is to select $H$ in a way that minimizes the "distance" between the Kernel Density Estimator $\hat{p}_H(\vec{x})$ and the true distribution $p(\vec{x})$. Most methods aim to minimize the *mean integrated square error* $\mathbb{E}\left[\int_{\vec{x}} \left(p(\vec{x}) - \hat{p}_H(\vec{x})\right)^2 d\vec{x}\right]$, which is just the expected $L_2$ loss between distributions $p$ and $\hat{p}_H$.

What makes this problem challenging is that the true distribution is generally assumed to be unknown [1]. Bandwidth selectors solve this problem by selecting $H$ based on an approximation of the true distribution. The most straightforward approach is to replace $p(\vec{x})$ by the Normal distribution $\mathcal{N}(\mu, \Sigma)$ with the observed mean $\mu$ and covariance matrix $\Sigma$ of the data sample. In this case, the optimal (diagonal) bandwidth matrix has a closed-form form expression that can be computed via *Scott's rule* [1]:

$$\hat{h}_i^{scott} = s^{-\frac{1}{d+4}} \cdot \sigma_i \tag{3}$$

In this formula, $s$ denotes the sample size, $d$ the dimensionality, and $\sigma_i$ the standard deviation of the $i$-th attribute in the sample. However, since real-world data is seldom normal, Scott's rule often leads to overly smoothed estimators that underfit the true distribution and can produce arbitrarily large estimation errors [14, 23].

More sophisticated selection methods follow the same general principle, but apply better methods to approximate $p(\vec{x})$. In particular, the two most prominent classes of bandwidth selectors are *Cross-Validation* methods, which use leave-one-out cross-validation based on the data sample to approximate $p(\vec{x})$ [5, 37], and *Plug-In* methods, which iteratively refine a pilot distribution for $p(\vec{x})$ that was plugged into the bandwidth optimization [45]. While these methods produce drastically better bandwidth estimates than Scott's rule, they are also expensive to compute, making their usage hard to justify in time-critical settings.

The bandwidth selection problem for KDE-based methods has also been covered in the database literature. However, while all KDE-related publications acknowledge the problem, most simply rely on Scott's rule due to its simplicity [14, 15, 16, 18, 39]. In fact, there are only three papers that discuss alternatives: In [4], Blohfeld et. al. demonstrate that a Plug-In method can drastically improve the selectivity estimation performance of a one-dimensional KDE model. Instead of relying on an existing method, Zhang et. al. introduce a Newton-based optimization method in [46] that directly minimizes the skyline cardinality estimation error based on a sampled test set from the database. While their method is conceptually similar to the one introduced in this paper, it only discusses the case of a single bandwidth parameter that is used for all dimensions. Finally, in [2], Andrzejewski et. al. demonstrate how to reduce the performance impact of a Cross-Validation bandwidth selector by using GPU-acceleration.

## 3.3 Bandwidth Selection in a Database

Most bandwidth selection methods have one thing in common: They are built on the assumption that the true distribution $p(\vec{x})$ is unknown. In the case of a selectivity estimator, we can relax this assumption since we have a clear and measurable objective: Our estimator is used to approximate the dataset that is stored in the database. Accordingly, we should pick the bandwidth in a way that minimizes the estimation error between our model and the database.

Exploiting this observation, we can re-formulate the bandwidth selection problem for a KDE-based selectivity estimator into the following constrained optimization problem:

$$H^* = \underset{H}{\operatorname{argmin}}\, \mathbb{E}\left[\mathcal{L}\left(\hat{p}_H(\Omega), \frac{|\sigma_{\vec{x}\in\Omega}(R)|}{|R|}\right)\right] \quad \text{s.t. } \forall i : h_i > 0 \tag{4}$$

In this formula, $|R|$ denotes the cardinality of relation $R$ and $|\sigma_{\vec{x}\in\Omega}(R)|$ denotes the number of tuples from $R$ that fall into the region $\Omega$. Summarizing the optimization problem, we want to pick the (positive) bandwidth that minimizes the expected value of a given loss function $\mathcal{L} : \mathbb{R}^2 \to \mathbb{R}$ over the range queries on relation $R$. The actual choice of loss function is arbitrary, typical choices are quadratic (L2), absolute (L1), or relative. Appendix Section C.1 provides a list of commonly used error metrics and explains how to apply them to this method.

In order to solve optimization problem (4), we need to compute the expected estimation error across all possible range queries on $R$. Since this is obviously infeasible to compute, we will instead solve a tractable approximation. In particular, we can approximate the expected error by averaging the estimation error over a small set of representative range queries. This method yields an asymptotically correct approximation and can be efficiently computed. In other words, given a training set of range queries $Q = \{\Omega_1, \ldots, \Omega_q\}$, we obtain the optimal bandwidth by solving the following modified optimization problem:

$$H^* = \underset{H}{\operatorname{argmin}} \frac{1}{q} \sum_{i=1}^{q} \mathcal{L}\left(\hat{p}_H\left(\Omega_i\right), \frac{|\sigma_{\vec{x}\in\Omega_i}\left(R\right)|}{|R|}\right) \quad (5)$$
$$\text{s.t. } \forall i : h_i > 0$$

We derive the closed-form expression for the gradient $\nabla_H \mathcal{L}$ of the error function with respect to the bandwidth in Appendix Section C. Based on this, we can compute the optimal bandwidth by simply plugging the estimation error and its gradient into any off-the-shelf optimization algorithm for bound constrained problems.

There are two issues to consider: First, the problem is non-convex, meaning it can have multiple local minima. It is therefore important to use a global optimization algorithm when solving the problem. However, it should be noted that we found that the actual number of minima is very low, typically only one or two. Second, finding the optimal bandwidth requires a viable set of representative range queries. One possibility is to use a random workload, which would result in the model being optimized under the assumption that every region is equally likely to be queried. However, in real-world scenarios this is rarely the case, as there are usually regions that the users query more frequently. Accordingly, the model optimization should prioritize estimation quality in these regions, which can be achieved by collecting a set of user queries and solving problem (5) for them.

## 3.4 An Optimal KDE Selectivity Estimator

Summarizing our approach, we can construct an optimal KDE-based selectivity estimator for a given relation $R$ by the following three steps:

1. Collect a set of representative queries on relation $R$, e.g. by keeping the last $q$ user queries in a ring buffer. Increasing $q$ leads to a more robust optimization, but at the same time makes the process more expensive. We found that a good choice for $q$ is on the order of a few hundred queries.

2. Collect a random sample of size $s$ from the target relation and initialize the bandwidth using Scott's rule from equation (3). Increasing the sample size improves the estimation quality, but also makes it more expensive to compute. Therefore, $s$ should be chosen as large as possible, while still guaranteeing that the time to estimate a selectivity stays within a given time budget.

3. Pick the optimal bandwidth $H^*$ by plugging optimization problem (5) into your favorite gradient-based numerical solver for bound constrained optimization problems. The required closed-form expression for the gradient $\nabla_H \mathcal{L}$ is given in Appendix Section C.2. In our experiments, we made good experiences with first running a coarse global optimization algorithm (e.g. MLSLS [24]) to get us into the right neighborhood, followed by a local optimization algorithm (e.g. L-BFGS-B [8] or MMA [40]) to refine the bandwidth.

# 4. SELF-TUNING KDE MODELS

The method outlined in the previous section describes how to build an optimal KDE-based selectivity estimator. However, if either the database or the workload characteristics change, we have to rebuild the estimator to remain optimal. Obviously, this is not ideal, as periodic rebuilding could incur significant performance overhead. Ideally, the estimator should adjust itself to compensate for any occurring changes. In this section, we present two complementary methods to build such a self-tuning KDE model.

**Listing 1:** Adaptively adjusting the bandwidth.

```
1  g⃗^(0) = [0,...,0]^T  //Gradient accumulator.
2  m⃗ = [0,...,0]^T  //Running average magnitude.
3  λ⃗ = [1,...,1]^T  //Initial learning rates.
4  i = 0; t = 0
5  foreach query Ω:
6     Compute p̂_H(Ω) according to (2).
7     Run query Ω.
8     Collect |σ_{x⃗∈Ω}(R)| from query feedback.
9     g⃗^(t) = g⃗^(t) + ∇_H L(p̂_H(Ω), |σ_{x⃗∈Ω}(R)|/|R|)
10    i = i + 1
11    if (i mod N == 0):
12       g⃗^(t) = g⃗^(t)/N
13       foreach dimension d:
14          m_d = α·m_d + (1−α)·(g_d^(t))^2
15          if (g_d^(t)·g_d^(t−1) > 0)  λ_d = min(λ_d·λ_inc, λ_max)
16          if (g_d^(t)·g_d^(t−1) < 0)  λ_d = max(λ_d·λ_dec, λ_min)
17          h_d = max(0.5·h_d, h_d − λ_d/√m_d · g_d^(t))
18       t = t + 1; g⃗^(t) = [0,...,0]^T
```

## 4.1 Adaptive Bandwidth Maintenance

Changes in the query workload and to the database can lead to a gradual change in the optimal bandwidth configuration. In order to counter these changes, we propose to continuously update the bandwidth by incrementally solving optimization problem (5) via a variant of *stochastic gradient descent* (SGD). The principle idea is simple: Instead of optimizing the bandwidth based on a set of pre-collected user queries, we directly update the bandwidth after each incoming query by subtracting the gradient $\nabla_H \mathcal{L}$ of our error function. This leads to a reactive optimization procedure that automatically adjusts itself to changes.

There are some issues to consider: First, SGD is susceptible to outliers since extreme gradients from single observations can lead to strong bandwidth fluctuations between queries. We mitigate this problem by using a *mini-batch* approach, where we average the gradients from a small number of queries before updating the model. Second, the convergence behavior of SGD is controlled by the *learning rate*, which determines how strongly each observation changes the model. This rate should tend towards zero for static workloads to accelerate convergence, but increase again when the workload changes for quick and reactive model updates. In our estimator we use *RMSprop* [42], which is the mini-batch variant of the earlier *Rprop* [36], to achieve this behavior. RMSprop changes the learning rate based on the direction of previous gradients: If the directions of the last gradients agree, the rate is increased, otherwise it is decreased. Before updating the model, RMSprop also scales the gradients by the average magnitude of recent gradients to achieve better convergence behavior. RMSprop is a simple and cheap learning algorithm that we found to work well in our experiments. Finally, we need to ensure that the positivity constraint from optimization problem (5) is never violated. SGD is an unconstrained optimization algorithm, meaning we have to manually guarantee positivity. We do this by artificially restricting model updates towards zero to at most half the current bandwidth's value.

Listing 1 gives a detailed overview of this adaptive bandwidth learning algorithm: When a new query arrives, we compute the selectivity estimate (line 6), pass the estimate to the database, and let the query run (line 7). After the query finished, we receive query feedback, compute the error gradient according to Appendix Section C.2, and add it to the current mini-batch (line 9). If the mini-batch is full, we average the accumulated gradient (line 12) and use it to update the running average of gradient magnitudes (line 14). We then perform the RMSprop learning rate update for each dimension (lines 15 and 16), and update the individual bandwidth parameters by subtracting the scaled gradients (line 17), enforcing the positivity constraint as discussed.

There are a few important parameters in this algorithm: The mini-batch size $N$ controls how many gradients are averaged per mini-batch. We found that a value around 10 works well. The smoothing rate $\alpha$ limits the influence of historic gradients on the gradient scaling. In our implementation, this is set to 0.9. Parameters $\lambda_{min}$ and $\lambda_{max}$ are the smallest and largest allowed learning rates. They are set to $10^{-6}$ and 50 respectively, which are the suggested values from [42]. The final two parameters are $\lambda_{inc}$ and $\lambda_{dec}$, which are the multiplicative factors by which the learning rate is in-/decreased. These are also set to their suggested values from [42], which are 1.2 and 0.5 respectively.

## 4.2 Karma-based Sample Maintenance

The second major source of estimation errors are updates to the database: These updates invalidate our data sample, forcing us to apply corrective measures to keep it representative. In our case, this is an especially challenging problem, since the sample resides on the graphics card, while database updates occur on the host. Keeping sample and database in sync via traditional sample maintenance algorithms [12, 33] would require us to continuously transfer information to the graphics card. Depending on the workload, these additional transfers could easily clog the limited bandwidth of the PCI Express bus, resulting in severe performance penalties. We therefore had to develop a novel sample maintenance algorithm that avoids data transfers where possible.

The first scenario we consider are insert-only workloads. In this case, we utilize the well-known *reservoir sampling algorithm* [43]. Reservoir sampling adds newly inserted data to the sample with probability $|S|/|R|$, replacing a random point in the process. It is optimal with regard to transfers, as all decisions are made independently by the host and only points that will end up in the sample are transferred to the graphics card.

Things get more complicated for workloads that also contain deletions or updates, as reservoir sampling is insufficient in this case. In our estimator, we use an approximate, but transfer-efficient, maintenance algorithm. Our method piggybacks on the query feedback information that is sent to the graphics card for the adaptive bandwidth optimization. It uses this information to identify and selectively replace points that consistently hurt our estimation quality. In order to derive the approach, let us recall equation (2): The selectivity estimate $\hat{p}_H(\Omega)$ is computed by averaging the individual contributions $\hat{p}_H^{(i)}(\Omega)$ from all points in the sample. We can now remove the $i$-th point's contribution from the estimate:

$$\hat{p}_H^{-(i)}(\Omega) = \frac{\hat{p}_H(\Omega) \cdot s - \hat{p}_H^{(i)}(\Omega)}{s-1} \tag{6}$$

This *adjusted estimate* $\hat{p}_H^{-(i)}(\Omega)$ tells us how the estimator would have performed if the $i$-th point had been absent. Our key assumption is, that if a point significantly improves estimation quality by its absence, it is likely outdated and should be replaced. In order to track this, we introduce the *Karma score*, which we define as the estimation error change a given point causes for a given query $\Omega$:

$$K^{(i)}(\Omega) = \left[\mathcal{L}\left(p(\Omega), \hat{p}_H^{-(i)}(\Omega)\right) - \mathcal{L}\left(p(\Omega), \hat{p}_H(\Omega)\right)\right] \tag{7}$$

Large positive Karma values correspond to sample points that significantly improved the estimation quality in query region $\Omega$, while negative values are associated with points that had a negative impact. By aggregating the Karma over multiple queries, we obtain a solid indicator to tell us which points consistently helped and which hurt us. Accordingly, we define the $i$-th point's *cumulative Karma* as follows:

$$K_{t+1}^{(i)} = \min\left(K_t^{(i)} + K^{(i)}(\Omega), K_{\max}\right) \tag{8}$$

Based on this equation, the estimator updates the cumulative Karma for all points after each query. If a point's Karma falls below a given threshold, it is marked as outdated and is replaced by a newly sampled point from the database. It should be noted that – strictly speaking –, this approach is not a true maintenance algorithm, as changes in the database are not applied instantaneously to the sample. Instead they propagate over time as the Karma of deleted points decays. We play two tricks to reduce this sample convergence time. First, we limit the maximum cumulative Karma that a point can collect through the saturation constant $K_{\max}$ in equation (8)[3]. Second, we use a shortcut to quickly identify and replace sample points that fall into an empty query region. Details of this shortcut are explained in Appendix Section E.

# 5. GPU-ACCELERATED KDE MODELS

In a prior publication, we introduced the idea of tapping into the massive raw computational power of modern consumer graphics cards to enable more advanced methods for selectivity estimation [16]. KDE models are especially well-suited for GPU-acceleration: All major operations – including estimation, model optimization, and sample maintenance – can be expressed as summations over individual probability contributions from sampled points. Computing these is an embarrassingly parallel problem that can be efficiently accelerated by a GPU (or any other massively parallel processor for that matter).

Figure 3 illustrates the general implementation overview of our self-tuning estimator, the required operations, and the relations between them. Our implementation is integrated into Postgres 9.3.1 and uses OpenCL to achieve portable performance across a multitude of devices, including modern GPUs and multi-core CPUs [17]. In the following sections, we discuss the single components in more detail. However, for lack of space, we will not provide a highly detailed rundown of our implementations. Instead, we convey the general ideas, and refer to the provided source code for further details. We also assume that the reader is familiar with the kernel-based programming model that is typically used for GPU programming.

[3]We found that a good value for $K_{\max}$ is four.

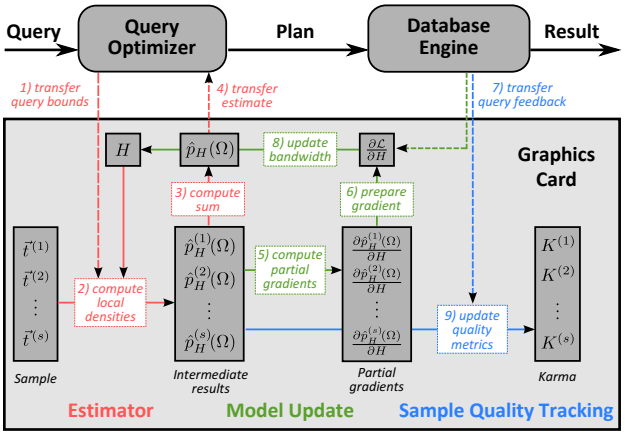



**Figure 3:** Computing the selectivity on a graphics card.

## 5.1 Data Representation

The most memory-intensive part of our estimator is the sample buffer, which is used to keep the sampled data points on the graphics card. We represent the sample in row-major format, using floating point numbers to store attribute values. While a columnar data layout would likely be preferable from a performance perspective [6, 17], the row-major format allows us to efficiently update points in the sample using only a single PCI Express transfer.

In order to reduce the performance impact of the row-major format, we use templated kernel code and runtime code generation: When an estimator is initialized for a table, we compile specific kernels for it, passing both the configured floating point precision and the table's dimensionality as compile-time constants to the OpenCL runtime compiler. This helps the runtime compiler to apply device-specific code optimizations like loop-unrolling or data access reordering.

## 5.2 Initializing the Estimator

Model construction is triggered by Postgres' ANALYZE command. When called, we utilize Postgres' internal routines to collect a random sample of the requested size. We copy this sample row-by-row into a host buffer, transforming data types where necessary. This buffer is then copied over to a pre-allocated sample buffer on the graphics card. Note that this is the only major data transfer that is required by the estimator! Besides preparing the data sample, we also initialize the estimator bandwidth via Scott's rule from equation (3). The most expensive part of this formula is to compute the standard deviation within the dimensions: We efficiently compute this on the graphics card by summing up the single dimensions' values and their squares in a parallel binary reduction scheme [19], and then using the identity: $\sigma^2 = {}^1\!/_n \sum_j x_i^2 - \left({}^1\!/_n \sum_i x_i\right)^2$.

## 5.3 Optimizing the Bandwidth

In order to optimize the bandwidth based on a query workload as described in Section 3.4, we first transfer the query bounds and the observed selectivities of the workload to the graphics card. The actual optimization of problem (5) then happens in two steps: We first run a global optimization step via MLSL [24] to (hopefully) get us near the global minimum, followed by a local optimization via L-BFGS-B [8] to refine the bandwidth. Note that we did not implement these optimization algorithms ourselves. Instead, we plug the optimization problem into off-the-shelf implementations from the optimization library NLopt [22]: During every internal iteration, NLopt calls one of our functions to compute the estimation error and the gradient for the current bandwidth. Internally, this function copies the current bandwidth to the graphics card and then lets each thread compute the partial gradient and estimation error for a single user query according to equation (17). Afterwards, the partial gradients are summed up using a parallel binary reduction scheme and shipped back to he host, where the final gradient is returned to NLopt, which uses it to update the bandwidth according to the selected optimization algorithm. After a few iterations, NLopt will converge to the optimal bandwidth, which we then transfer to the graphics card.

## 5.4 Computing Estimates

The actual estimator is mostly implemented as outlined in [16]: First, we transfer the query bounds $\Omega$ to the device (1). Based on these bounds, and the current bandwidth $H$, the graphics card computes the local densities in parallel, with each thread using equation (13) to compute the individual contribution $\hat{p}_H^{(i)}(\Omega)$ of a single sample item (2). The local contributions are written to a temporary buffer, which is afterwards aggregated via a parallel binary reduction scheme (3). The resulting estimate $\hat{p}_H(\Omega)$ is transferred back to the host (4), where it is passed on to the query optimizer. The only major difference to our previous implementation is that we do not discard the temporary buffer that stores the individual contributions until after the query returns from the database, as we need them for the sample maintenance.

## 5.5 Model Updates

The adaptive bandwidth optimization method outlined in Section 4.1, requires us to compute the gradient of the current estimation error after each query, which is quite expensive to do. Luckily, we can hide these costs behind the query runtime. As derived in Appendix Section C, the gradient is the product of two factors: The first one is the partial derivative of the estimator with respect to the bandwidth, as given by equation (15). This part is expensive to compute, but is independent of the actual estimation error, meaning we can compute it on the graphics card while the database is busy running the query. The computation itself is parallelized over the sample items, with each thread using equation (16) to compute the partial gradient contribution for one sample point (5). Afterwards, all partial contributions are aggregated using a parallel binary reduction scheme (6). The second factor is the partial derivative of the estimation error, which is a simple scalar that is computed from the estimated and the true selectivity. After we receive the true selectivity (7), the host computes this factor according to equation (14) and ships it to the graphics card. There it is multiplied with the pre-computed model-dependent factor, and accumulated on the current mini-batch. Once a full mini-batch is collected, we update the bandwidth (8) as described in Listing 1.

Another implementation aspect we did not cover so far are logarithmic bandwidth updates. In our experiments, we found that updating the logarithm of the bandwidth often leads to improved estimates and a more stable optimization behaviour. In fact, we observed improvements over the non-logarithmic case in 68% of all experiments. Appendix Section D provides further details about this method.

## 5.6 Sample Maintenance

The final major component is the sample maintenance algorithm, which consists of two components: Reservoir Sampling is used to deal with insertions and Karma-based Sample Maintenance to deal with updates.

Our implementation of Reservoir Sampling follows the algorithm specification from [43]: Whenever a new tuple is inserted into relation $R$, the sample maintenance routine gets notified by the database engine. The host then decides whether the new tuple should be inserted into the sample as explained in [43]. If the tuple should be inserted, a PCI Express transfer is scheduled to copy it to a random position in the sample, overwriting the point that is stored there.

Karma-based Sample Maintenance tracks the cumulative Karma as described in Section 4.2 based on the (retained) individual contributions $\hat{p}_H^{(i)}(\Omega)$, the estimated, and the true selectivity. This computation happens in a single pass over the sample, with each thread evaluating equation (6) for one point. Afterwards, the same thread updates the point's cumulative Karma according to equation (8) (9). If this update results in a point's Karma falling below a threshold – or if the true selectivity is zero, and condition (20) holds –, the point is deemed outdated and will be replaced. The actual replacement happens in a two-step procedure: First, a bitmap is generated in parallel that indicates which tuples must be replaced. This bitmap is transferred back to the host, which samples the requested number of tuples from the database and schedules transfers to copy them to the positions indicated in the bitmap.

# 6. EVALUATION

In this section, we present the experimental evaluation of our proposed estimator. The experiments were designed to cover the following four areas: 1) Estimation quality on static datasets, 2) Estimation behavior with increasing model size, 3) Performance impact of our method & advantage of GPU-acceleration, and 4) Estimation quality on changing datasets.

## 6.1 Experimental Setup

We will now describe the datasets, workloads, and estimators that were used for the experiments.

### 6.1.1 Compared Estimators

We compared the following five estimators:

**KDE heuristic:** The "naïve" variant of a Kernel Density Estimator, using Scott's rule (c.f. equation (3)) to compute the bandwidth. This is our baseline to compare against existing KDE estimators.

**KDE SCV:** A Kernel Density Estimator using the bandwidth chosen by a state-of-the-art bandwidth selection method. In particular, we use the bandwidth selector *Hscv.diag* from the R package *ks*[4], which is based on the Smoothed Cross Validation approach from [11].

**STHoles:** STHoles [7] is a self-tuning, multidimensional histogram that continuously refines its model based on query feedback. We use STHoles as a proxy to compare our estimator against the quality of state-of-the-art multidimensional histograms.

**KDE batch:** The optimal KDE estimator, as described in Section 3. In this configuration, we optimize the bandwidth during model construction based on an initial training set of user queries.

**KDE adaptive:** The adaptive variant of our estimator, as described in Section 4. In this configuration, we initialize the model according to Scott's rule, and then continuously adjust the bandwidth based on query feedback. We also use the Karma-based sample maintenance techniques.

### 6.1.2 Evaluated Datasets

We based our experiments on a variety of synthetic and real-world[5] datasets from multiple domains to get a solid and unbiased understanding of the estimation quality:

**Bike:** Hourly aggregated usage statistics for the Washington DC bike sharing system. The dataset consists of 17,379 data points with 16 continuous attributes.

**Forest:** Geological survey of forest cover types in the US. The dataset consists of 581,012 points with 54 attributes, of which we use a projection on the 10 continuous ones.

**Power:** Time series describing the electric power consumption in a single household with one-minute resolution. The dataset consists of 2,075,259 data points with 9 attributes containing continuous and discrete values.

**Protein:** Physiochemical properties of the tertiary structure of proteins. The dataset contains 45,730 points with 9 continuous attributes.

**Synthetic:** Synthetic dataset from [14], consisting of one million points. It is generated by randomly placing hyper-rectangular clusters with a uniform interior distribution, and then adding uniformly distributed noise.

For all datasets, we generated versions with three and eight dimensions. These were created by projecting the full dataset onto a random subset of the available attributes.

### 6.1.3 Query Workloads

We generated our workloads according to the method outlined in [7]: Each workload is specified by a distribution for the query centers and a target measure that the queries have to meet. Based on this approach, we defined the following four workloads, which cover a variety of usage scenarios:

**DT:** Queries with a target selectivity of 1% whose centers are following the data distribution. This workload corresponds to a set of well-defined user queries that return roughly the same number of tuples.

**DV:** Queries with a target volume of 1% of the data space whose centers are following the data distribution. This workload corresponds to a set of explorative user queries having a wide spectrum of different selectivities.

**UT:** Queries with a target selectivity of 1%, whose centers are randomly distributed across the data space. This workload corresponds to a random workload with queries having highly diverse query volumes.

**UV:** Queries with a target volume of 1% of the data space, whose centers are randomly distributed across the data space. This workload corresponds to a random workload with mostly empty queries.

[4] `http://cran.r-project.org/web/packages/ks`

[5] Obtained from the UCI Machine Learning repository [3].

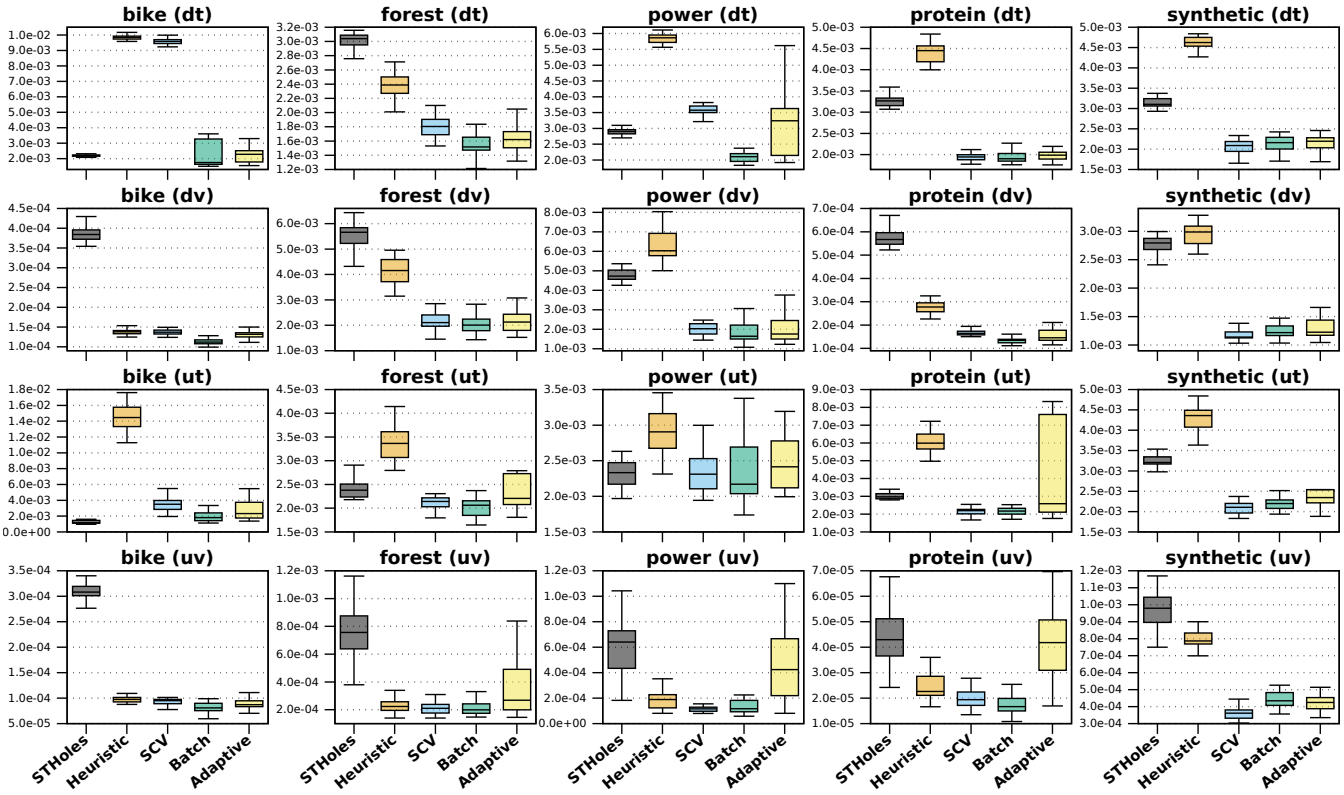



**Figure 4:** Estimation quality on static datasets (3D). The y-axis shows the absolute selectivity estimation error.

## 6.2 Estimation Quality on Static Data

In the first series of experiments, we compared the estimation quality on static datasets with fixed workloads. The primary goal of these experiments was to answer the following three questions:

1. Does selecting the bandwidth by solving optimization problem (5) from Section 3 actually improve the estimation quality of a KDE-based selectivity estimator, especially when compared to using Scott's rule, or alternative bandwidth selection methods?
2. How does the adaptive bandwidth optimization from Section 4 compare against the batch method?
3. How does the estimation quality of the batch-optimized and the adaptive estimator compare against state-of-the art multidimensional histograms?

We ran these experiments using the following protocol: We randomly selected 100 training and 300 test queries from the selected workload. Then, we initialized the estimators, and – if applicable – optimized their model parameters based on the training queries. Finally, we measured the average absolute selectivity estimation error on the test set. This process was repeated 25 times to measure how strongly the estimation quality fluctuates. During each run, all estimators were given the exact same set of queries to avoid measuring workload-dependent changes. Furthermore, all KDE-based estimators were built using the same random sample. Finally, in order to make the comparisons fair, we restricted all estimators to use the same amount of memory. In particular, we allowed $d \cdot 4\,\text{kB}$, where $d$ is the dimensionality of the dataset.

| | STHoles | Heuristic | SCV | Batch | Adaptive | All |
|---|---|---|---|---|---|---|
| **STHoles** | - | *52.5* | *19.4* | *14.4* | *24.5* | *8.4* |
| **Heuristic** | *46,0* | - | *5.7* | *6.7* | *12.1* | *0.7* |
| **SCV** | *79.0* | *94.3* | - | *37.0* | *46.3* | *26.3* |
| **Batch** | *84.1* | *90.8* | *63.0* | - | *64.5* | *43.2* |
| **Adaptive** | *71.3* | *81.8* | *53.5* | *35.5* | - | *21.4* |

**Table 1:** Comparing the estimation quality of the different estimators: Cells list the percentage of experiments in which the row's estimator performed better than the one on top.

The results of these experiments are shown in Figure 4 for the three-, and in Figure 5 for the eight-dimensional datasets. Each plot shows the result for a single workload and dataset, using boxplots to illustrate how estimation errors were distributed over the 25 repetitions. Furthermore, Table 1 lists the percentage of experiments in which a given estimator performed better than a given other one.

Let us discuss the results: First, the plots clearly show that our methods indeed substantially improve the estimation quality of KDE. In fact, *Batch* performed better than *Heuristic* in over 90% of all experiments! It should also be noted that *Batch* even outperformed *SCV* in a majority of cases ($> 60\%$). While this seems counter-intuitive at first, it is expected: *SCV* picks the bandwidth by minimizing the Cross Validation error on the sample. *Batch* has two advantages: It directly minimizes the selectivity estimation error, and – through the user workload – takes additional information about the data distribution into account, allowing it to better tune the bandwidth.

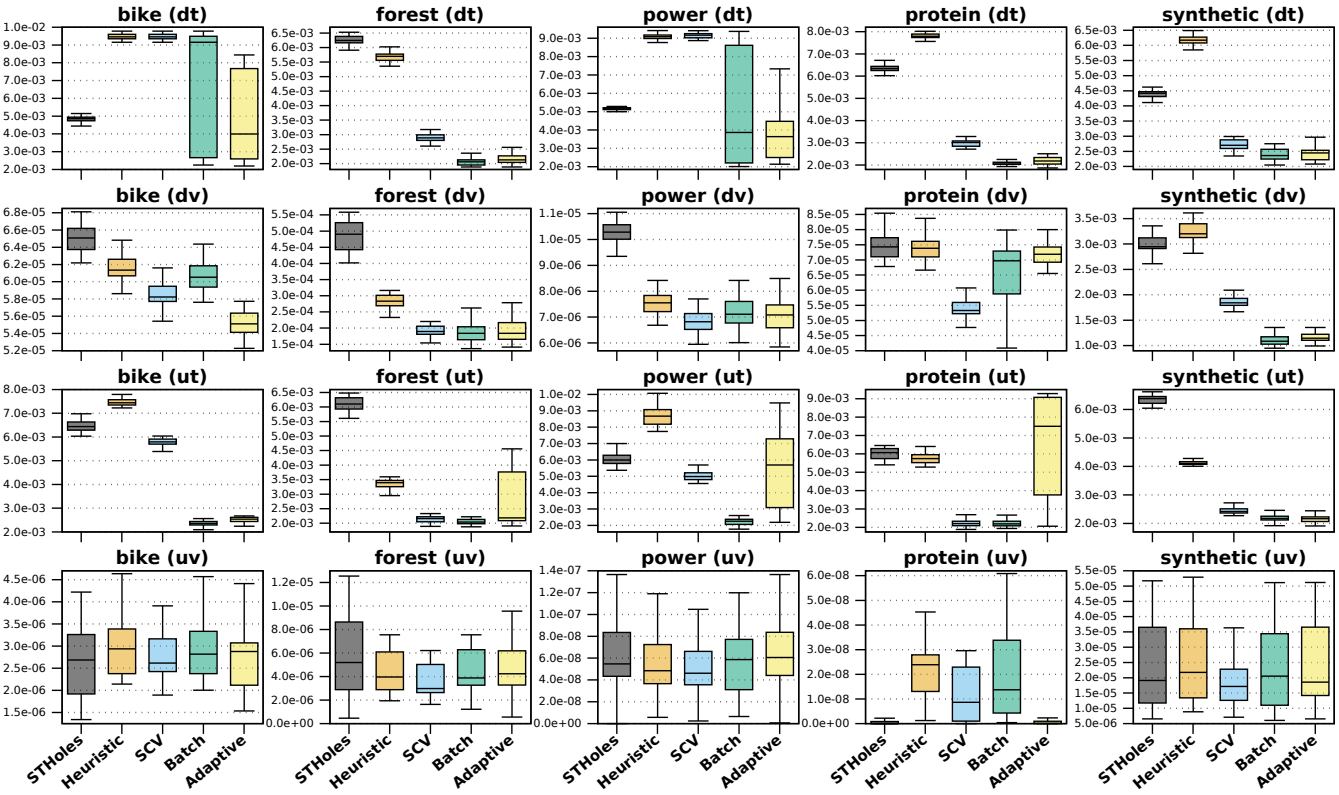



**Figure 5:** Estimation quality on static datasets (8D). The y-axis shows the absolute selectivity estimation error.

Second, while *Adaptive* does not fully match the estimation quality of *Batch*, it still performs admirably, clearly outperforming *Heuristic* and matching the quality of *SCV*. However, there are several cases in which *Adaptive* leads to a much higher variance than *Batch* or *SCV*. There are two primary reasons for this behavior: First, the stochastic nature of online learning leads to a higher variance in the produced bandwidth estimates. Second, in contrast to *Batch*, *Adaptive* does not use global optimization, meaning it can get stuck in local minima more easily. Still, *Adaptive* generally offers competitive performance, while not requiring us to collect user queries or run an expensive optimization routine.

Finally, let's compare with *STHoles*. In general, both *Batch* and *Adaptive* are clearly superior, with *Batch* producing better estimates in 84% and *Adaptive* in 71% of all experiments. However, *STHoles* still has the advantage of producing more stable estimates: Since bandwidth optimization is a non-convex problem, there is some chance of the estimator getting stuck in a local minima, leading to a large variance – as can be seen in Bike(8D) DT and Power(8D) DT. Nevertheless, we could demonstrate that our bandwidth selection methods clearly improve the estimation quality of a state-of-the-art KDE-based selectivity estimator, and push them into a range where KDE actually becomes a reasonable alternative to multidimensional histograms.

## 6.3 Quality Impact of Model Size

In this experiment, we investigated how the estimators' quality scales when increasing the model size. Note that we focused on *Heuristic*, *Batch* & *Adaptive* – a discussion of *STHoles'* scaling behavior can be found in [7].

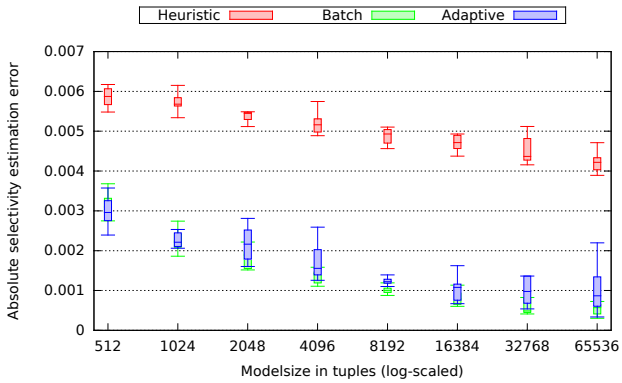



**Figure 6:** Estimation quality with growing model size.

The experiment was set up like the previous section, using the 8D Forest dataset and the DT workload. Estimators were built based on 100 randomly selected queries, the estimation quality – the absolute selectivity estimation error – was measured based on another 100 queries. Each experiment was repeated ten times, the results are illustrated in Figure 6, note the logarithmically scaled x-axis. The plots clearly show that, for all models, using larger samples leads directly to improved estimates: By increasing the sample size from 1024 to 32768, we could decrease the estimation error to roughly one third. In general, the error decreases exponentially with a negative exponent with increasing sample size. Furthermore, the experiment again demonstrates the impact of bandwidth optimization, as the optimized estimators are roughly twice as accurate as *Heuristic*.

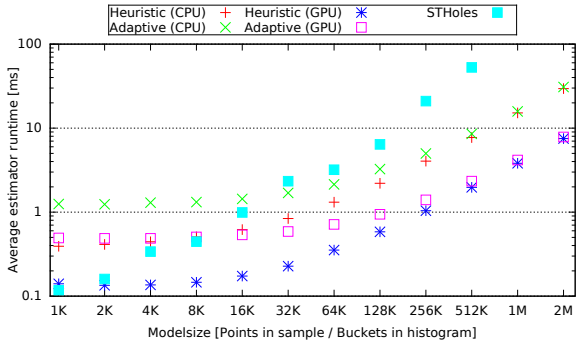



**Figure 7:** Estimator runtime with growing model size.

## 6.4 Performance & GPU-Acceleration

In the next experiment, we measured the estimator runtime overhead with growing model sizes, answering the following questions:

1. How does the runtime of our estimators scale with increasing model size?
2. What is the runtime impact of the adaptive bandwidth optimization method?
3. How much does GPU-acceleration help?
4. How does the runtime of our methods compare to state-of-the-art multidimensional histograms?

In this experiment, we measured the total estimation overhead for 100 random UV queries on a synthetic 8D table with three million rows. We compared the performance of *Heuristic*, *Adaptive* & *STHoles – Batch* was left out as it is as fast as *Heuristic*. For *Heuristic* and *Adaptive*, we measured both the CPU and the GPU variant. Measuring the runtime of STHoles was somewhat tricky, as it incrementally builds a histogram by adding buckets (or "holes") based on query feedback until a limit is reached. Therefore, the estimation speed of STHoles gradually decreases over time until the histogram is full. This is different from KDE, whose runtime overhead is constant and only depends on the model size. In order to compensate for this, we report the runtime overhead for the full STHoles model, which was constructed over a large-enough training workload. Furthermore, we only measured estimation time, excluding the time that STHoles spends on maintaining the histogram.

The experiment was run on custom server with a Quad-Core Intel Xeon E5620 @ 2.4GHz and 32GB of DDR3-1066 RAM, running Scientific Linux 6.4. It is equipped with a middle-class NVIDIA GTX-460 graphics card having 2GB of DDR4 memory. The GPU is controlled by NVIDIA's 340.32 driver, we used Intel's OpenCL SDK 2014 to run the estimators on the CPU. Figure 7 illustrates the results, note that both axes are scaled logarithmically.

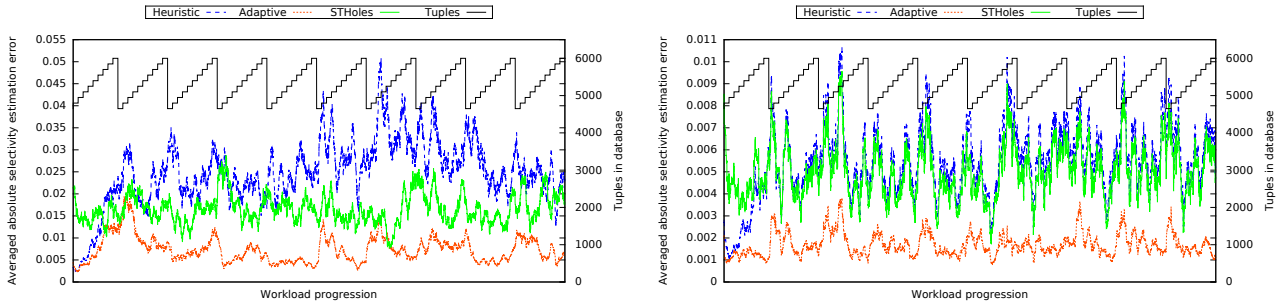
First, let us take a look at how estimation overhead scales with increasing model sizes. On both CPU & GPU, *Heuristic* and *Adaptive* show a flat curve until the model size reaches $\approx 16-32K$ points. Afterwards, the estimator scales linearly with the model size, as expected. The flat start is likely caused by the OpenCL runtime, which introduces latency penalties for each scheduled data transfer and kernel execution. Only for larger models is this latency shadowed by the actual computation. Second, let us discuss the runtime difference between *Adaptive* and *Heuristic*. Interestingly, this difference is constant and does not seem to depend on the model size. This behavior demonstrates nicely that efficiently hide the additional computations required by *Adaptive*: They run concurrently while Postgres executes the query, making the only measurable performance impact of *Adaptive* the latency penalties incurred by the additional kernel calls and data transfers.

Let us now discuss the performance impact of using a graphics card: Looking at the plot, we can see that GPU-acceleration gives us a speed-up of around a factor of four, both for *Heuristic* and for *Adaptive*. In particular, the GPU can estimate a selectivity with *Adaptive* on a model of $128K$ elements in under $1ms$. The CPU is slightly slower, but still delivers impressive performance, being able to estimate selectivities based on models of $\approx 32K$ elements in $1ms$. This is a nice indication of the portable performance of OpenCL, and demonstrates that our estimator is also very efficient when being run without a graphics card. This impressive scaling behaviour has some interesting consequences: In the first series of experiments, we restricted our samples to $32kB$ of memory. Judging from the performance measurements, we could easily increase this size to at least $2MB$ without seeing any major performance impact. Looking at the previous section, increasing the sample by a factor of 64 would cut our estimation error by at least a factor of three. This demonstrates one of the strongest advantages of KDE: Since they can be efficiently parallelized, KDE models scale much better, and can benefit much stronger from modern hardware than multidimensional histograms or comparable methods could.

Finally, let us compare *STHoles* and *KDE*. The plot shows that *STHoles* is generally faster than KDE for small models – due to the OpenCL overhead –, but gets much slower when the model size is increased. In fact, for large models, STHoles is around a factor of $7-10$ slower than KDE on the GPU, and around a factor of 3 than KDE on the CPU. Note however that this comparison is only indicative and should be taken with a grain of salt: First, we only measured the estimation component of *STHoles*, excluding the time spent on model maintenance. This is actually highly significant, given that *STHoles'* maintenance time can quickly add up to several seconds on larger models. Second, even though we implement all performance improvements outlined in [7], *STHoles* only uses a sequential implementation, meaning it has somewhat sub-optimal performance-

## 6.5 Estimation Quality on Changing Data

In the final experiment, we wanted to demonstrate the self-tuning aspects of *Adaptive*, in particular the Karma-based sample maintenance from Section 4.2. For this experiment, we used a synthetic workload, combining insertions, deletions, and selections. The workload starts by loading 4500 tuples, evenly distributed among three random clusters. Afterwards the workload features ten cycles of slowly creating a new cluster by gradually inserting 1500 tuples into it, followed by deleting all tuples belonging to one of the old clusters. These dataset changes are interleaved with a DT query workload that queries older clusters less frequently than newer ones. This scenario mirrors an evolving database where new data is queried more frequently, and older data is periodically moved into an archive. We ran this workload ten times each on *STHoles*, *Heuristic*, and *Adaptive*, measuring the progression of the absolute estimation error. As in the first series of experiments, we restricted the estimators to $d{\cdot}4$ kB of memory.




**(a)** 5D **(b)** 8D

**Figure 8:** Estimation quality on changing datasets.

Figure 8 illustrates the results of this experiment, with Figure 8a containing the 5D, and Figure 8b the 8D case. The black lines on top illustrate the progression of the number of tuples in the database. The observed behaviors are as expected: *Heuristic* is unable to keep up with the ongoing database changes. *STHoles* is somewhat in the middle: Due to its self-tuning nature, it does a better job of adjusting to the changes, but cannot compete with *Adaptive*.

# 7. CONCLUSION

In this paper, we proposed a modern multidimensional selectivity estimator based on Kernel Density Estimation (KDE). KDE is a statistical to estimate probability densities from a data sample that has several advantages over histograms. However, existing KDE-based selectivity estimators were a) suboptimal, because they did not adjust the important bandwidth parameter, b) non-adaptive, because they did not use sample maintenance routines, and c) expensive to evaluate. These shortcomings prevented KDE-based selectivity estimators from becoming a reasonable alternative to multidimensional histograms.

We tackled these challenges and significantly advanced the state-of-the-art in KDE-based selectivity estimation. First, we demonstrated how to numerically optimize the bandwidth based on query feedback to improve estimation quality. Then, we presented two orthogonal methods enabling self-tuning models based on query feedback: A learning algorithm to continuously adjust the bandwidth, and a transfer-efficient sample maintenance algorithm to identify and replace outdated sample points. Finally, we explained how these methods can be efficiently accelerated by graphics cards and multi-core CPUs, and provided a working prototype of the estimator integrated into Postgres. We validated our estimator in an extensive evaluation, demonstrating that its estimation quality outperforms state-of-the-art multidimensional histograms, that it is efficient to evaluate, can automatically adjust itself to database updates, and can be scaled up to very large model sizes without showing a significant performance impact.

# 8. FUTURE WORK

In order to conclude the paper, we now present and discuss selected topics that we believe to be interesting directions for future work in the field of KDE-based selectivity estimation.

*Join Selectivity Estimation.* An important subproblem in selectivity estimation research is how to estimate selectivities over joined tables. Extending KDE-based models for this case is very interesting. If the predicate is known beforehand – for instance in case of PK-FK joins –, it can be done by building the estimator based on a sample collected directly from the join result, e.g. by using the sampling algorithms presented in [9]. Estimating arbitrary theta joins is more complex: Since KDE defines a continuous estimator, we should in theory be able to express join selectivities by a joint integral over the two estimators. However, it remains to be seen whether such a formulation is a viable approach, and whether it can be computed efficiently.

*Support for Discrete and String Data.* KDE works best on continuous data that has strong neighborhood properties, i.e. where the presence of data increases the likelihood of finding other data. However, in real-world databases, we also have to deal with non-continuous data. Extending KDE-based selectivity estimators for discrete and string data is therefore very important. While we did not demonstrate this behavior in the evaluation, our estimator already handles discrete attributes to a certain degree: The bandwidth optimization will observe that it does not profit from increasing the bandwidth for discrete attributes and therefore set it to a very small value. Effectively, this means that the estimator automatically degrades to counting matching tuples on discrete attributes. However, this support is fairly limited, and further investigation is required. One starting point is statistics literature, where this topic was already studied in the context of KDE over mixed continuous and discrete variables [27].

*Variable Kernel Density Models.* Variable – or adaptive – KDE models are an extension of KDE using distinct bandwidth parameters for each sample point that can be individually modified [41]. These models have shown very promising results in density estimation for very high-dimensional spaces. In theory, our bandwidth optimization approach should be portable to variable KDE models as well, and it would be interesting to investigate how this additional complexity affects estimation quality and runtime overhead.

*Integrating with a GPU-accelerated DBMS.* Another interesting direction is integrating our estimator with a GPU-accelerated database system [6]. In this case, the estimator would have to compete with the query processor for resources on the GPU. This requires methods for resource sharing that can trade-off estimation quality with resource impact. In particular, using techniques such as *device fission*, modern graphics cards can be virtually partitioned into several sub-devices that can run code concurrently. This technique could allow a GPU-accelerated database system to allocate a given fraction of the graphics card – say 10% – for selectivity estimation without affecting query performance. It would also be interesting to investigate how to accelerate KDE estimation across multiple graphics cards.

# 9. REFERENCES


[1] *Multivariate Density Estimation - Theory, Practice and Visualization*. John Wiley & Sons, Inc., 1992.

[2] W. Andrzejewski, A. Gramacki, and J. Gramacki. Graphics processing units in acceleration of bandwidth selection for kernel density estimation. *Int. J. Appl. Math. Comput. Sci*, 23(4):869–885, 2013.

[3] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[4] B. Blohsfeld, D. Korus, and B. Seeger. A comparison of selectivity estimators for range queries on metric attributes. *SIGMOD Record*, 28(2):239–250, 1999.

[5] A. W. Bowman. An alternative method of cross-validation for the smoothing of density estimates. *Biometrika*, 71(2):353–360, 1984.

[6] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[7] N. Bruno, S. Chaudhuri, and L. Gravano. Stholes: a multidimensional workload-aware histogram. *SIGMOD Record*, 30(2):211–222, 2001.

[8] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[9] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. *SIGMOD Record*, 28(2):263–274, 1999.

[10] S. Christodoulakis. Implications of certain assumptions in database performance evauation. *ACM Transactions on Database Systems (TODS)*, 9(2):163–186, 1984.

[11] T. Duong and M. L. Hazelton. Cross-validation bandwidth matrices for multivariate kernel density estimation. *Scandinavian Journal of Statistics*, 32(3):485–506, 2005.

[12] R. Gemulla, W. Lehner, and P. J. Haas. Maintaining bounded-size sample synopses of evolving datasets. *The VLDB Journal*, 17(2):173–201, 2008.

[13] P. B. Gibbons, Y. Matias, and V. Poosala. Maintaining a random sample of a relation in a database in the presence of updates to the relation, Jan. 4 2000. US Patent 6,012,064.

[14] D. Gunopulos, G. Kollios, J. Tsotras, and C. Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, 2005.

[15] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. *ACM SIGMOD Record*, 29(2):463–474, 2000.

[16] M. Heimel and V. Markl. A first step towards gpu-assisted query optimization. In *International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures (ADMS), Istanbul, Turkey*, pages 33–44, 2012.

[17] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013.

[18] C. Heinz and B. Seeger. Cluster kernels: Resource-aware kernel density estimators over streaming data. *Knowledge and Data Engineering, IEEE Transactions on*, 20(7):880–893, 2008.

[19] D. Horn. Stream reduction operations for gpgpu applications. *Gpu gems*, 2:573–589, 2005.

[20] Y. Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th VLDB conference*, pages 19–30. VLDB Endowment, 2003.

[21] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. *SIGMOD Record*, 20(2):268–277, 1991.

[22] S. G. Johnson. The NLopt nonlinear-optimization package. `http://ab-initio.mit.edu/nlopt`.

[23] M. C. Jones, J. S. Marron, and S. J. Sheather. A brief survey of bandwidth selection for density estimation. *Journal of the American Statistical Association*, 91(433):401–407, 1996.

[24] A. R. Kan and G. Timmer. Stochastic global optimization methods part ii: Multi level methods. *Mathematical Programming*, 39(1):57–78, 1987.

[25] P.-A. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality estimation using sample views with quality assurance. In *Proceedings of the 2007 ACM SIGMOD Conference*, pages 175–186. ACM, 2007.

[26] J.-H. Lee, D.-H. Kim, and C.-W. Chung. Multi-dimensional selectivity estimation using compressed histogram information. *SIGMOD Record*, 28(2):205–214, 1999.

[27] Q. Li and J. Racine. Nonparametric estimation of distributions with categorical and continuous data. *journal of multivariate analysis*, 86(2):266–292, 2003.

[28] R. J. Lipton, J. F. Naughton, and D. A. Schneider. *Practical selectivity estimation through adaptive sampling*, volume 19. ACM, 1990.

[29] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. Consistent selectivity estimation via maximum entropy. *The VLDB journal*, 16(1):55–76, 2007.

[30] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. *SIGMOD Record*, 27(2):448–459, 1998.

[31] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment*, 2(1):982–993, 2009.

[32] M. Muralikrishna and D. J. DeWitt. Equi-depth multidimensional histograms. *SIGMOD Record*, 17(3):28–36, 1988.

[33] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *Data Engineering, 1992. Proceedings. Eighth International Conference on*, pages 632–641. IEEE, 1992.

[34] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd VLDB conference*, pages 486–495. VLDB Endowment, 1997.

[35] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st VLDB conference*, pages 1228–1239. VLDB Endowment, 2005.

[36] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[37] S. R. Sain, K. A. Baggerly, and D. W. Scott. Cross-validation of multivariate densities. *Journal of the American Statistical Association*, 89(427):807–817, 1994.

[38] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. Isomer: Consistent histogram construction using query feedback. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 39–39. IEEE, 2006.

[39] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. In *Proceedings of the 32nd VLDB conference*, pages 187–198. VLDB Endowment, 2006.

[40] K. Svanberg. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM journal on optimization*, 12(2):555–573, 2002.

[41] G. R. Terrell and D. W. Scott. Variable kernel density estimation. *The Annals of Statistics*, pages 1236–1265, 1992.

[42] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.

[43] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[44] M. Wand and M. Jones. Comparison of smoothing parameterizations in bivariate kernel density estimation. *Journal of the American Statistical Association*, 88(422):520–528, 1993.

[45] M. Wand and M. Jones. Multivariate plug-in bandwidth selection. *Computational Statistics*, 9(2):97–116, 1994.

[46] Z. Zhang, Y. Yang, R. Cai, D. Papadias, and A. Tung. Kernel-based skyline cardinality estimation. In *Proceedings of the 2009 ACM SIGMOD Conference*, pages 509–522. ACM, 2009.

# APPENDIX

## A. PRELIMINARIES

In the following, we provide an overview of the math behind various interesting aspects of KDE-based selectivity estimator. In particular, we will derive the closed-form expression that is used to compute a KDE range estimate based on a data sample, derive the gradient of the estimation error with regard to the bandwidth, explain how to perform logarithmic bandwidth updates, and introduce a shortcut to quickly identify which sample points fall into a given query region. We demonstrate these topics for a Gaussian Kernel, which has the following form:

$$K\left(\vec{x}\right) = \left(2\pi\right)^{-\frac{d}{2}} \exp -\frac{1}{2}\vec{x}^T\vec{x} \tag{9}$$

Note that we use the Gaussian Kernel primarily to simplify the following derivations. In principle, our methods can be applied to any differentiable kernel function. Another interesting choice is for instance the *Epanechnikov Kernel*, which is a truncated second-degree polynomial. Compared to the Gaussian, the Epanechnikov Kernel is much cheaper to compute. However, its limited support make the derivation more cumbersome.

## B. KDE FOR RECTANGULAR REGIONS

Equation (2) defines how to compute the KDE-based selectivity estimate $\hat{p}_H\left(\Omega\right)$ for a given query region $\Omega$. In order to evaluate this equation, we need to derive a closed-form expression for the individual probability mass contribution $\hat{p}_H^{(i)}\left(\Omega\right)$ from a given sample point $\vec{t}^{(i)}$:

$$\hat{p}_H^{(i)}\left(\Omega\right) = \int_\Omega \frac{1}{|H|} K\left(H^{-1}\left[\vec{t}^{(i)} - \vec{x}\right]\right) d\vec{x} \tag{10}$$

In general, the integral in this term doesn't have a closed form solution for arbitrary regions [1]. However, since we assume that $\Omega$ is rectangular, i.e. $\Omega = (l_1, u_1) \times \ldots \times (l_d, u_d)$, we can integrate by each dimension individually:

$$p_H^{(i)}(\Omega) = \int_{l_1}^{u_1} \ldots \int_{l_d}^{u_d} \frac{K\left(H^{-1}\left[\vec{t}^{(i)} - \vec{x}\right]\right)}{|H|} dx_d \ldots dx_1 \tag{11}$$

We can now simplify (11) via the following observations:

1. The assumption of a diagonal bandwidth matrix $H = \operatorname{diag}\left(h_1, \ldots, h_d\right)$ allows us to simplify the computation of the determinant $|H|$ to: $\prod_{i=1}^d h_i$. We can also express the product $H^{-1}\left[\vec{t}^{(i)} - \vec{x}\right]$ in terms of single dimensions: $\left[\frac{1}{h_1}\left(x_1 - t_1^{(i)}\right), \ldots, \frac{1}{h_d}\left(x_d - t_d^{(i)}\right)\right]^T$.
2. The multivariate Gaussian Kernel is a product kernel [1], meaning it can be expressed as a product of single-dimensional kernels: $K\left(\vec{x}\right) = \prod_{i=1}^d K\left(x_i\right)$. In particular, this allows us to evaluate the $d$-dimensional integral in (11) as a product of $d$ one-dimensional integrals.

Plugging the definition of the Gaussian Kernel from (9) into (11) and applying the discussed simplifications, we arrive at:

$$p_H^{(i)}(\Omega) = \prod_{j=1}^d \int_{l_j}^{u_j} \underbrace{\frac{\left(2\pi\right)^{-\frac{1}{2}}}{h_j} \exp\left(-\frac{\left(x_j - t_j^{(i)}\right)^2}{2 \cdot h_j^2}\right)}_{=\mathcal{N}_{t_j^{(i)}, h_j^2}\left(x_j\right)} dx_j \tag{12}$$

In equation (12), $\mathcal{N}_{\mu,\sigma^2} : \mathbb{R} \to \mathbb{R}$ denotes the *probability density function* of the one-dimensional normal distribution with variance $\sigma^2$ and mean $\mu$. We can now get rid of the integral by plugging the *cumulative distribution function* of the normal distribution into equation (12):

$$\begin{aligned} p_H^{(i)}(\Omega) &= \prod_{j=1}^d \frac{1}{2}\left[\operatorname{erf}\left(\frac{u_j - t_j^{(i)}}{\sqrt{2} \cdot h_j}\right) - \operatorname{erf}\left(\frac{l_j - t_j^{(i)}}{\sqrt{2} \cdot h_j}\right)\right] \\ &= \frac{1}{2^d} \prod_{j=1}^d \left[\operatorname{erf}\left(\frac{u_j - t_j^{(i)}}{\sqrt{2} \cdot h_j}\right) - \operatorname{erf}\left(\frac{l_j - t_j^{(i)}}{\sqrt{2} \cdot h_j}\right)\right] \end{aligned} \tag{13}$$

Where $\operatorname{erf} : \mathbb{R} \to \mathbb{R}$ denotes the *error function*, which is defined as $\operatorname{erf}\left(x\right) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-\tau^2} d\tau$.

# C. GRADIENT OF THE LOSS FUNCTION

In order to numerically solve optimization problem (5) with a gradient-based solver, we need to derive the gradient $\nabla_H \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial h_1}, \ldots, \frac{\partial \mathcal{L}}{\partial h_d}\right]^T$ of the chosen loss function $\mathcal{L}$ with respect to the bandwidth $H$. Looking at the partial derivative with respect to $h_i$ and applying the chain rule, we arrive at:

$$\frac{\partial \mathcal{L}}{\partial h_i} = \frac{\partial \mathcal{L}}{\partial \hat{p}_H(\Omega)} \cdot \frac{\partial \hat{p}_H(\Omega)}{\partial h_i} \tag{14}$$

According to this equation, we can compute each element of the gradient as the product of two partial derivatives. The first derivative is independent of the estimator and encodes information about the chosen loss function $\mathcal{L}$. The second derivative encodes how the estimator reacts when we change the bandwidth of the $i$-th attribute. We will now take a closer look at both factors individually.

## C.1 Partial Derivative of the Loss Function

Let us first take a look at the partial derivative $\frac{\partial \mathcal{L}}{\partial \hat{p}_H(\Omega)}$ of the loss function. This partial derivative encodes information about the chosen loss function. In particular, it specifies how the selected error function behaves when the estimated selectivity changes. By changing this factor, we can modify which error metric the model optimization will minimize, allowing us to fine-tune the bandwidth of our KDE-based estimator with regard to any differentiable error metric. The following list gives loss functions and closed-form partial derivatives for several important metrics:

- Quadratic (L2) Error:

$$\mathcal{L}_{\text{Quadratic}} = \left(\hat{p}_H(\Omega) - p(\Omega)\right)^2$$
$$\frac{\partial \mathcal{L}_{\text{Quadratic}}}{\partial \hat{p}_H(\Omega)} = 2 \cdot \left(\hat{p}_H(\Omega) - p(\Omega)\right)$$

- Absolute (L1) Error:

$$\mathcal{L}_{\text{Absolute}} = \left|\hat{p}_H(\Omega) - p(\Omega)\right|$$
$$\frac{\partial \mathcal{L}_{\text{Absolute}}}{\partial \hat{p}_H(\Omega)} = \begin{cases} -1 & \hat{p}_H(\Omega) < p(\Omega) \\ 0 & \hat{p}_H(\Omega) = p(\Omega) \\ 1 & \hat{p}_H(\Omega) > p(\Omega) \end{cases}$$

- Relative Error[6]:

$$\mathcal{L}_{\text{Relative}} = \frac{\left|\hat{p}_H(\Omega) - p(\Omega)\right|}{\lambda + p(\Omega)}$$
$$\frac{\partial \mathcal{L}_{\text{Relative}}}{\partial \hat{p}_H(\Omega)} = \frac{1}{\lambda + p(\Omega)} \cdot \begin{cases} -1 & \hat{p}_H(\Omega) < p(\Omega) \\ 0 & \hat{p}_H(\Omega) = p(\Omega) \\ 1 & \hat{p}_H(\Omega) > p(\Omega) \end{cases}$$

- Squared Relative Error[5]:

$$\mathcal{L}_{\text{Relative}^2} = \left(\frac{\hat{p}_H(\Omega) - p(\Omega)}{\lambda + p(\Omega)}\right)^2$$
$$\frac{\partial \mathcal{L}_{\text{Relative}^2}}{\partial \hat{p}_H(\Omega)} = 2 \cdot \frac{\hat{p}_H(\Omega) - p(\Omega)}{\lambda + p(\Omega)}$$

- Squared Q-Error[5] [31]:

$$\mathcal{L}_{\text{Q}^2} = \left[\log\left(\lambda + \hat{p}_H(\Omega)\right) - \log\left(\lambda + p(\Omega)\right)\right]^2$$
$$\frac{\partial \mathcal{L}_{\text{Q}^2}}{\partial \hat{p}_H(\Omega)} = 2 \cdot \frac{\log\left(\lambda + \hat{p}_H(\Omega)\right) - \log\left(\lambda + p(\Omega)\right)}{\lambda + \hat{p}_H(\Omega)}$$

## C.2 Partial Derivative of the Estimator

The second partial derivative from equation (14) encodes the model-specific part of the gradient. In particular, it specifies how strongly the estimators' selectivity estimate changes when we modify the bandwidth of the $i$-th attribute. We derive the closed-form expression for this factor by applying the sum and the constant factor derivation rule to the definition of the estimator from equations (2) and (13):

$$\frac{\partial \hat{p}_H(\Omega)}{\partial h_i} = \frac{1}{2^d \cdot s} \sum_{j=1}^{s} \frac{\partial}{\partial h_i} \prod_{k=1}^{d} \left[\operatorname{erf}\left(\frac{u_k - t_k^{(j)}}{\sqrt{2} \cdot h_k}\right) - \operatorname{erf}\left(\frac{l_k - t_k^{(j)}}{\sqrt{2} \cdot h_k}\right)\right] \tag{15}$$

We now apply the product rule to further simply the equation:

$$\frac{\partial \hat{p}_H(\Omega)}{\partial h_i} = \frac{1}{2^d \cdot s} \sum_{j=1}^{s} \frac{\partial}{\partial h_i} \left[\operatorname{erf}\left(\frac{u_i - t_i^{(j)}}{\sqrt{2} \cdot h_i}\right) - \operatorname{erf}\left(\frac{l_i - t_i^{(j)}}{\sqrt{2} \cdot h_i}\right)\right] \prod_{k \neq i} \left[\operatorname{erf}\left(\frac{u_k - t_k^{(j)}}{\sqrt{2} \cdot h_k}\right) - \operatorname{erf}\left(\frac{l_k - t_k^{(j)}}{\sqrt{2} \cdot h_k}\right)\right] \tag{16}$$

Applying the chain rule and observing that $\frac{d}{dx} \operatorname{erf}\left(\frac{c}{x}\right) = -\frac{2 \cdot c}{\sqrt{\pi} \cdot x^2} \cdot \exp\left(-\left(\frac{c}{x}\right)^2\right)$, we arrive at the closed-form formula to compute the partial derivative of the estimator with regard to the bandwidth of the $i$-th attribute:

$$\frac{\partial \hat{p}_H(\Omega)}{\partial h_i} = \frac{\sqrt{2}}{\sqrt{\pi} \cdot h_i^2 \cdot 2^d \cdot s} \cdot \sum_{j=1}^{s} \left[\left(l_i - t_i^{(j)}\right) \cdot \exp\left(-\frac{\left(l_i - t_i^{(j)}\right)^2}{2 \cdot h_i^2}\right) - \left(u_i - t_i^{(j)}\right) \cdot \exp\left(-\frac{\left(u_i - t_i^{(j)}\right)^2}{2 \cdot h_i^2}\right)\right] \prod_{k \neq i} \left[\operatorname{erf}\left(\frac{u_k - t_k^{(j)}}{\sqrt{2} \cdot h_k}\right) - \operatorname{erf}\left(\frac{l_k - t_k^{(j)}}{\sqrt{2} \cdot h_k}\right)\right] \tag{17}$$

# D. LOGARITHMIC MODEL UPDATES

As mentioned in Section 5.5, we found in our experiments that we can improve stability and quality of the adaptive bandwidth optimization routine by scaling the gradient to update the logarithm of the bandwidth. In order to perform these logarithmic updates, we need to slightly modify the gradient from Section C. In particular, if we define

[6] $\lambda$ is a small positive smoothing constant to prevent divisions by zero.

$h_i^* = \ln h_i$, we need to compute the modified partial gradient $\partial\mathcal{L}/\partial h_i^*$ to update the model. We can easily derive this gradient by applying the chain rule and a few simple transformations:

$$\frac{\partial\mathcal{L}}{\partial h_i^*} = \frac{\partial\mathcal{L}}{\partial h_i} \cdot \frac{\partial h_i}{\partial h_i^*} = \frac{\partial\mathcal{L}}{\partial h_i} \cdot \frac{\partial \exp h_i^*}{\partial h_i^*} = \frac{\partial\mathcal{L}}{\partial h_i} \cdot \exp h_i^* = \frac{\partial\mathcal{L}}{\partial h_i} \cdot h_i \tag{18}$$

In other words, we can directly optimize the logarithmic bandwidth by scaling each dimension of the gradient by the (non-logarithmic) value of the bandwidth. Besides scaling the gradient, we need to make another small change to the adaptive learning algorithm outlined in Listing 1: We have to remove the positivity safeguard from line 17, since it would prevent negative logarithmic bandwidths, and thus bandwidth values smaller than one.

# E. IDENTIFYING POINTS IN A REGION

As discussed in Section 4.2, our estimator identifies outdated sample points by tracking their *Karma score*, i.e. their net contribution to the estimation quality. While this approach is very efficient with regard to the required data transfers, it also results in the sample lagging behind the database, as updates only propagate over time. One shortcut to reduce this convergence time is to exploit when the system receives a query over a region $\Omega_0 = (l_1, u_1) \times \ldots \times (l_d, u_d)$ for which the database does not contain any tuples. Obviously, all sample points falling into $\Omega_0$ are outdated and can be instantly replaced by newly sampled points, vastly accelerating sample convergence in this region. In order to do this efficiently, we need to quickly identify which points from the sample are falling into a given region, for instance by scanning the sample.

Another option, that does not require a full scan of the sample, is to examine the individual probability contributions $p_H^{(i)}(\Omega_0)$. The basic idea is simple: A point from within the query region will typically produce a higher probability contribution than a point from outside of it. If we can bound the maximum probability that a point from outside of $\Omega_0$ can contribute, any point with a higher contribution must necessarily be from within $\Omega_0$. While this approach will not identify all points from within $\Omega_0$, we can piggyback the computations while updating the Karma Scores, making it very cheap to run.

Since the Gaussian Kernel is unimodal and symmetric, the theoretically maximum probability contribution from points outside of the query region is 0.5. However, this bound is only tight if the bandwidth is very small compared to the query region. In order to get a general tight bound, we first need to derive the largest possible probability contribution from points *within* $\Omega_0$. Obviously, the point $\vec{t}_{(0)} = \left(\frac{l_1+u_1}{2}, \ldots \frac{l_d+u_d}{2}\right)$, which is directly at the center of $\Omega_0$, will produce this contribution. Inserting $\vec{t}_{(0)}$ into equation (13), we arrive at:

$$p_H^{\mathbf{max}}(\Omega_0) = \prod_{j=1}^{d} \operatorname{erf}\left(\frac{u_j - l_j}{2 \cdot \sqrt{2} \cdot h_j}\right) \tag{19}$$

Based on this, we can easily derive the maximum probability contribution from a point outside of $\Omega_0$ by moving $\vec{t}_{(0)}$ along one of the dimensions until we hit the boundary. For example, if we move along dimension $j$, we arrive at $\vec{t}_{(j)} = \left(\frac{l_1+u_1}{2}, \ldots, l_j, \ldots, \frac{l_d+u_d}{2}\right)$. This ensures that the probability contributions from all but one dimensions remain at maximum, while the contribution from the $j$-th dimension is the largest possible for a point just outside of $\Omega_0$. Now, since the estimator uses different bandwidth parameters for each dimension, we will receive different contributions for $\vec{t}_{(1)}$ to $\vec{t}_{(d)}$. Out of those $d$ contributions, the largest one is the bound we are looking for.

Summarizing the approach, we can guarantee that point $\vec{t}^{(i)}$ is contained within query region $\Omega$ if the following condition holds:

$$p_H^{(i)}(\Omega) \geq \frac{p_H^{\mathbf{max}}(\Omega)}{2} \cdot \max_{j=1}^{d} \frac{\operatorname{erf}\left(\frac{u_j - l_j}{\sqrt{2} \cdot h_j}\right)}{\operatorname{erf}\left(\frac{u_j - l_j}{2 \cdot \sqrt{2} \cdot h_j}\right)} \tag{20}$$


## Acknowledgments

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) under funding code 01IS12056. Furthermore, we thank the anonymous reviewers for their helpful comments in preparing the paper's final version.