Quality and Efficiency in Kernel Density Estimates for Large Data*

Yan Zheng, Jeffrey Jestes, Jeff M. Phillips, Feifei Li
School of Computing, University of Utah, Salt Lake City, USA
{yanzheng, jestes, jeffp, lifeifei}@cs.utah.edu

ABSTRACT

Kernel density estimates are important for a broad variety of applications. Their construction has been well-studied, but existing techniques are expensive on massive datasets and/or only provide heuristic approximations without theoretical guarantees. We propose randomized and deterministic algorithms with quality guarantees which are orders of magnitude more efficient than previous algorithms. Our algorithms do not require knowledge of the kernel or its bandwidth parameter and are easily parallelizable. We demonstrate how to implement our ideas in a centralized setting and in *MapReduce*, although our algorithms are applicable to any large-scale data processing framework. Extensive experiments on large real datasets demonstrate the quality, efficiency, and scalability of our techniques.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management – *Systems*

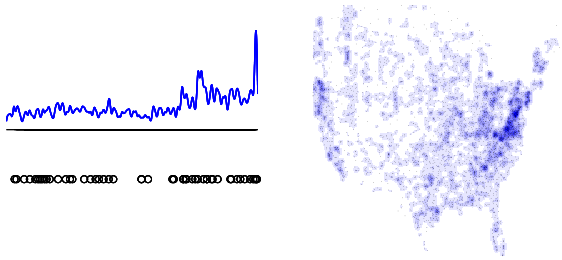
Keywords

Kernel density estimate (KDE), distributed and parallel KDE

1. INTRODUCTION

A kernel density estimate is a statistically-sound method to estimate a continuous distribution from a finite set of points. This is an increasingly common task in data analysis. In many scientific computing and data intensive applications, the input data set P is a finite number of observations or measurements made for some real-world phenomena, that can be best described by some random variable V with an unknown probability distribution function (pdf) f. For example, temperature readings from sensor nodes, collected over a period of time, represent a finite sample of a 1-dimensional random variable *temperature*.

Given a data set P consisting of values from a domain $\mathcal{M}$, a kernel density estimate is a function f_P that for any input in $\mathcal{M}$ (not necessarily in P) describes *the density* at that location. It is a fundamental data smoothing problem where inferences about the population are made, based on a finite data sample. That said, we view P as a finite, independent and identically distributed (iid) data sample drawn from a random variable V that is governed by an unknown distribution f. We are interested in estimating the shape of this function f. The kernel density estimate f_P approximates the density of f at any possible input point $x \in \mathcal{M}$ [31, 37]. Figure 1 visualizes the kernel density estimate (KDE) in both 1 and 2 dimensions, using real data sets (a web trace in 1D and a spatial dataset from openstreetmap in 2D). Black dots represent a subset of points from P, and the blue curves or regions represent the KDE constructed from P.



(a) KDE in 1D. (b) KDE in 2D.

Figure 1: Kernel density estimate (KDE).

A kernel density estimate has useful and important applications in databases. Suppose we have a set of customers labeled by their annual income P. Then we may ask do we have more customers with income about 100K or income about 50K (rather than specifying a rigid range). A kernel density estimate is a function f_P that for any input (such as $100K$ or $50K$) describes *the density* at that location. So we could return the values $f_P(100K)$ and $f_P(50K)$ to compare.

Alternatively, we may have a database storing the (two-dimensional) locations of crimes in a database, and we may want to compare the density of these crimes around two houses (that we may be considering buying). In this case, rigid notions of districts may not appropriately capture the density at houses near the borders of these districts.

Another family of applications originates from statistical physics where the KDE value is proportional to the force in a system of particles. In this setting, very high precision answers are needed to accurately carry out simulations.

Our contributions. A kernel density estimate serves two main purposes. First, it is a data structure that can process function evaluation queries. Second, it is a summarization, like a histogram, for quickly presenting the distribution of density over a domain. We will allow approximate kernel

*Thanks to support from NSF grants IIS-0916488, IIS-1053979, and CCF 1115677.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

density estimate queries; most large data summarizations are by nature approximations, and density evaluations are models and thus should not be used to arbitrary precision. This approximation allows us to focus on a trade-off between the size of the data structure and the allowed approximation error; we focus on the strong ℓ_∞ error between the original kernel density estimate and its approximation.

Moreover, unlike histograms, kernel density estimates generalize naturally to more than one dimensional data. Since many spatial data sets lie naturally in $\mathbb{R}^2$, we focus primarily on one and two-dimensional data; however our techniques will scale naturally to higher dimensions; see Section 5.6.

With these goals in mind we investigate how to calculate such a data structure on an enormous scale (our experiments use 100 million points); our approximation (with theoretical bounds) is quite small in size (requires only megabytes in space) and is in fact independent of the original number of points, and offers very accurate queries. More importantly, our methods for computing these approximations are designed with parallel and distributed processing in mind, and we demonstrate how to adapt them in the popular MapReduce framework. But they can be easily adopted by any other parallel and distributed computation models.

The paper is organized as follows. Section 2 presents the problem formulation. Section 3 discusses background material and related work and Section 4 discusses the one dimension case. Section 5 presents the two dimension case. Section 6 reports experimental results and Section 7 concludes. Some technical proofs are relegated to the Appendix.

2. PROBLEM FORMULATION

Let P be an input point set of size n. We will either consider $P \in \mathbb{R}^1$ or $P \in \mathbb{R}^2$. We first describe the situation for $\mathbb{R}^d$, and then specify to one of these two common cases.

We restrict the *kernels* $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$ we consider to satisfy the following three properties:

(K1) **mass preservation:** For any $p \in \mathbb{R}^d$, then
$$\int_{x \in \mathbb{R}^d} K(p, x)dx = 1.$$

(K2) **shift- and rotation-invariance:** There exists a function $\kappa : \mathbb{R}^+ \to \mathbb{R}^+$ such that for any $p, x \in \mathbb{R}^d$ we have $K(p, x) = \kappa(\|p - x\|)$, where $\|p - x\|$ denotes the ℓ_2 distance between p and x.

(K3) **monotonicity:** For $z < z'$ then $\kappa(z) > \kappa(z')$.

Examples of kernels include (described here for $\mathbb{R}^2$):

- Gaussian: $K(p, x) = \frac{1}{2\pi\sigma^2} \exp\left(\frac{-\|p-x\|^2}{2\sigma^2}\right)$
- Triangle: $K(p, x) = \frac{3}{\pi\sigma^2} \max\left\{0, 1 - \frac{\|p-x\|}{\sigma}\right\}$
- Epanechnikov: $K(p, x) = \frac{2}{\pi\sigma^2} \max\left\{0, 1 - \frac{\|p-x\|^2}{\sigma^2}\right\}$
- Ball: $K(p, x) = \begin{cases} 1/\pi\sigma^2 & \text{if } \|p - x\| < \sigma \\ 0 & \text{otherwise.} \end{cases}$

We use the Gaussian kernel by default (the most widely used kernel in the literature); although some scenarios favor the Epanechnikov kernel [39, 42]. All kernel definitions have a σ term to controls the amount of data smoothing. In this paper we assume that σ is fixed; however, choosing the appropriate σ is an important problem and there is a large literature on doing so [23, 35, 44]. A notable property of our main techniques is that *they do not require knowledge of the kernel type or the value of* σ, it is only used in the evaluation step (although bounds are worse with the Ball kernel; details will appear in the extended version of this paper).

Given such a kernel K and a point set P in d-dimensions a *kernel density estimate* is formally defined as a function KDE_P which for *any query* $x \in \mathbb{R}^d$ evaluates as
$$\text{KDE}_P(x) = \frac{1}{|P|} \sum_{p \in P} K(p, x). \quad (1)$$

However, in many data-intensive applications, P could be large, e.g., sensor readings from numerous sensors over a long period of time. Thus, evaluating a kernel density estimate over P directly, which takes $O(n)$, can be prohibitively expensive in big data. Hence, our goal is to construct another point set Q (often $Q \subset P$, but not necessarily) so that KDE_Q approximates KDE_P well. More precisely,

Definition 1 (Approximate Kernel Density Esimate) Given the input P, σ, and some error parameter $\varepsilon > 0$, the goal is to produce a set Q to ensure
$$\max_{x \in \mathbb{R}^d} |\text{KDE}_P(x) - \text{KDE}_Q(x)| = \|\text{KDE}_P - \text{KDE}_Q\|_\infty \leq \varepsilon, \quad (2)$$
and we call this an *ε-approximation*.

We can also allow some failure probability, in which case an additional parameter $\delta \in (0, 1)$ is introduced, and the goal is to produce Q to ensure
$$\Pr[\|\text{KDE}_P - \text{KDE}_Q\|_\infty \leq \varepsilon] \geq 1 - \delta, \quad (3)$$
and we call this an *(ε, δ)-approximation*.

Lastly, in some kernel density estimates, Q does not have to be a point set. Rather, it can be any succinct structure that approximates $\text{KDE}_P(x)$ well according to (2) or (3). □

Our objective is to find an approximate KDE_Q such that for any query x, evaluating $\text{KDE}_Q(x)$ is much cheaper than evaluating $\text{KDE}_P(x)$ which takes $O(n)$, and at the same time, $\text{KDE}_Q(x)$ approximates $\text{KDE}_P(x)$ well even in the worst case.

Often it will be useful to instead generate a weighted point set Q so there exists a weight $w : Q \to \mathbb{R}$ associated with every point in Q and in which case the kernel density estimate over Q is defined as:
$$\text{KDE}_Q(x) = \sum_{q \in Q} w(q)K(q, x). \quad (4)$$

3. BACKGROUND AND RELATED WORK

Around the 1980s kernel density estimates became the defacto way in statistics to represent a continuous distribution from a discrete point set [42] with the study initiated much earlier [37]. However, this work often implied brute force ($O(n)$ time) solutions to most queries.

The problem of evaluating kernel density estimates is a central problem in statistical physics and numerical analysis. These problems are often posed as n-body simulations where the force-interactions between all pairs of points need to be calculated [1], and the pairwise force is up to a constant described by the Gaussian kernel. This has resulted in many indexing type techniques that up to constant precisions can evaluate the kernel density estimate at all n points in roughly $O(n)$ time. These techniques are sometimes called *fast multi-pole methods* [4], and in practice these are typically implemented as quad trees which calculate the distance to roots of subtrees instead of all pairs

when the distance becomes far enough. Numerical approximation techniques called the (Improved) Fast Gauss Transform (IFGT) [14, 36, 49, 50] can further improve these approaches. But the IFGT approach (in general fast multipole methods) is based on heuristics and does not offer formal theoretical guarantees on the approximation-time trade-off.

In order to have a formal theoretical guarantee to derive an (ε, δ)-approximation, random sampling is a baseline method, but it requires $O(\frac{1}{\varepsilon^2} \log \frac{1}{\delta})$ samples to be included in Q, which could lead to expensive query evaluations especially for small ε and/or δ values.

A recent technique using discrepancy theory [33] creates a small representation of a kernel density estimate (smaller than the random sampling approach) while still bounding the ℓ_∞ error. It works by creating a min-cost matching of points in P; that is P is decomposed into $|P|/2$ pairs so that the sum over all distances between paired points is minimized. Then it randomly removes one point from each pair reducing the size of P by half. This process is repeated until either the desired size subset or the tolerable error level is reached. However, computing the min-cost matching [11] is expensive, so this approach is only of theoretical interest and not directly feasible for large data. Yet, this will serve as the basis for a family of our proposed algorithms.

A powerful type of kernel is a *reproducing kernel* [2, 32] (an example is the Gaussian kernel) which has the property that $K(p, q) = \langle p, q \rangle_{\mathcal{H}_K}$; that is, the similarity between objects p and q defines an inner-product in a *reproducing kernel Hilbert space* (RKHS) $\mathcal{H}_K$. This inner-product structure (the so-called "kernel trick"), has led to many powerful techniques in machine learning, see [38, 40] and references therein. Most of these techniques are not specifically interested in the kernel density estimate; however, the RKHS offers the property that a single element of this space essentially represents the entire kernel density estimate. These RKHS approximations are typically constructed through some form of random sampling [41,48], but one technique known as "kernel herding" [7] uses a greedy approach and requires significantly smaller size in theory, however it bounds only ℓ_2 error as opposed to the sampling techniques which bound a stronger ℓ_∞ error [24].

Kernel density estimates have been used in the database and data mining community for density and selectivity estimations, e.g., [17,51]. But the focus in these works is how to use kernel density estimates for approximating range queries and performing selectivity estimation, rather than computing approximate kernel density estimates for fast evaluations. When the end-objective is to use a kernel density estimate to do density or selectivity estimation, one may also use histograms [16, 22, 26, 34] or range queries [12, 13, 19, 20, 47] to achieve similar goals, but these do not have the same smoothness and statistical properties of kernel density estimates [42]. Nevertheless, the focus of this work is on computing approximate kernel density estimates that enable fast query evaluations, rather than exploring how to use kernel density estimates in different application scenarios (which is a well-explored topic in the literature).

4. WARM-UP: ONE DIMENSION

Efficient construction of approximate kernel density estimates in one-dimension is fairly straightforward. But it is still worth investigating these procedures in more detail since to our knowledge, this has not been done at truly large scale before, and the techniques developed will be useful in understanding the higher dimensional version.

Baseline method: random sampling (RS). A baseline method for constructing an approximate kernel density estimate in one dimension is random sampling. It is well known that [7, 33] if we let Q be a random sample from P of size $O((1/\varepsilon^2)\log(1/\delta))$ then with probability at least $1-\delta$ the random sample Q ensures that $\|\text{KDE}_P - \text{KDE}_Q\|_\infty \leq \varepsilon$.

That said, the first technique (RS) follows from this observation directly and just randomly samples $O((1/\varepsilon^2)\log(1/\delta))$ points from P to construct a set Q. In the centralized setting, we can employ the one pass reservoir sampling technique [46] to implement RS efficiently. For large data that is stored in distributed nodes, RS can still be implemented efficiently using the recent results on generating random samples from distributed streams [9].

The construction cost is $O(n)$. The approximate KDE has a size $O((1/\varepsilon^2)\log(1/\delta))$, and its query cost (to evaluate $\text{KDE}_Q(x)$ for any input x) is $O((1/\varepsilon^2)\log(1/\delta))$.

RS can be used as a preprocessing step for any other technique, *i.e.*, for any technique that constructs a KDE over P, we run that technique over a random sample from P instead. This may be especially efficient at extremely large scale (where $n \gg 1/\varepsilon^2$) and where sampling can be done in an efficient manner. This may require that we initially sample a larger set Q than the final output to meet the approximation quality required by other techniques.

Grouping selection (GS). A limitation in RS is that it requires a large sample size (sometimes the entire set) in order to guarantee a desired level of accuracy. As a result, its size and query cost becomes expensive for small ε and δ.

Hence, we introduce another method, called the *grouping selection* (GS) method. It leverages the following lemma, known as the γ-perturbation.

Lemma 1 *Consider n arbitrary values $\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ such that $\|\gamma_i\| \leq \gamma$ for each $i \in [n]$. Then let $Q = \{q_1, q_2, \ldots, q_n\}$ such that $q_i = p_i + \gamma_i$ for all $p_i \in P$. Then $\|\text{KDE}_P - \text{KDE}_Q\|_\infty \leq \gamma/\sigma$.*

PROOF. This follows directly from the $(1/\sigma)$-Lipschitz condition on kernel K (which states that the maximum gradient of K is $(1/\sigma)$), hence perturbing all points by at most γ affects the average by at most γ/σ. □

Using Lemma 1, we can select one point q in every segment ℓ of length $\varepsilon\sigma$ from P and assign a weight to q that is proportional to the number of points from P in ℓ, to construct an ε-approximate KDE of P. Specifically, GS is implemented as follows. After sorting P if it is not already sorted, we sweep points from smallest to largest. When we encounter p_i, we scan until we reach the first p_j such that $p_i + \varepsilon\sigma < p_j$. Then we put p_i (or the centroid of p_i through p_{j-1}) in Q with weight $w(p_i) = (j-i)/n$. Since Q constructed by GS is weighted, the evaluation of $\text{KDE}_Q(x)$ should follow the weighted query evaluation as specified in equation 4.

Theorem 1 *The method GS gives an ε-approximate kernel density estimate of P.*

PROOF. The weighted output of GS Q corresponds to a point set Q' that has $w(q)$ unweighted points at the same location of each $q \in Q$; then $\text{KDE}_Q = \text{KDE}_{Q'}$. We claim that Q' is an $\varepsilon\sigma$-perturbation of P, which implies the theorem.

To see this claim, we consider any set $\{p_i, p_{i+1}, \ldots, p_{j-1}\}$ of points that are grouped to a single point $q \in Q$. Since

all of these points are within an interval of length at most $\varepsilon\sigma$, each $p_{i+\ell}$ is perturbed to a distinct point $q'_{i+\ell} \in Q$ (at location q) that is at distance $\gamma_{i+\ell} \leq \varepsilon\sigma$. □

GS's construction cost is $O(n)$ if P is sorted, or $O(n \log n)$ otherwise. Its query cost is $O(|Q|)$, which in the worst case could be $|Q| = |P|$. And Q may be large depending on how densely points in P are co-located and the values of ε and σ. However, GS can be used as a post-processing step on top of any other method, such as using GS over the output of RS. This takes little overhead if the points are already sorted, such as in the output of SS (see below).

Sort-selection (SS). Our best method (SS) offers an ε-approximate kernel density estimate using only $O(\frac{1}{\varepsilon})$ samples in one-dimension, a significant improvement over random sampling. It leverages the following interesting result:

Lemma 2 *Consider a one dimensional sorted point set $P = \{p_1, p_2, \ldots, p_n\}$ where $p_i \leq p_{i+1}$ for all i. Let $P_j = \{p_i \in P \mid (j-1)\varepsilon n < i \leq j\varepsilon n\}$ for integer $j \in [1, \lceil 1/\varepsilon \rceil]$ such that $P = \cup P_j$. Then for any $Q = \{q_1, q_2, \ldots, q_{\lceil 1/\varepsilon \rceil}\}$ such that each $q_j \in P_j$ then $\|\text{KDE}_P - \text{KDE}_Q\|_\infty \leq 2\varepsilon$.*

If each $q_j = p_{\lceil (j-1/2)\varepsilon n \rceil}$, then $\|\text{KDE}_P - \text{KDE}_Q\|_\infty \leq \varepsilon$.

We now can construct the SS method based on Lemma 2. It simply selects $p_{\lceil (j-\frac{1}{2})\varepsilon n \rceil}$ from P into Q for each $j \in [1, \lceil 1/\varepsilon \rceil]$. This requires that P is sorted, and this can be done efficiently at very large scale using external merge sort. However, we can do better. Note that ε-approximate quantiles for $\lceil 1/\varepsilon \rceil$ quantile values are sufficient to construct Q, and we can use an efficient streaming or distributed algorithm for computing these ε-approximate quantiles [8,15,21, 27,28]. In particular, we only need to find the $\frac{\varepsilon n}{2}$th, $\frac{3\varepsilon n}{2}$th, $\frac{5\varepsilon n}{2}$th, …, $(n - \frac{\varepsilon n}{2})$th quantile values from P. And it is easy to verify that ε-approximations of these quantiles are sufficient to establish the 2ε result in Lemma 2.

Using the ε-approximate quantiles, SS has a construction cost of $O(n \log \frac{1}{\varepsilon})$; otherwise its construction cost is $O(n \log n)$. In either case, its size is only $O(\frac{1}{\varepsilon})$ and its query cost is also just $O(\frac{1}{\varepsilon})$.

4.1 Efficient Evaluation

Once we have obtained a set Q above so KDE_Q approximates KDE_P, we need to efficiently answer queries of $\text{KDE}_Q(x)$ for any $x \in \mathbb{R}$. The first obvious choice is a brute force computation (BF) where we directly calculate $\frac{1}{|Q|}\sum_{q \in Q} K(q, x)$. This has little overhead and its cost is obviously $O(|Q|)$. It is most efficient if $|Q|$ is particular small.

A second approach (MP) is to use the one-dimensional variant of multi-pole methods. We build a B-tree (or binary tree if Q fits in memory) on Q. Each node v will store the weight (or count) w_v of all nodes in the subtree and the smallest v_s and largest v_l coordinates of the subtree. We traverse the tree as follows, starting at the root. If $x \in [w_s, w_l]$ visit each child and return their sum to the parent. If $|K(v_s, x) - K(v_l, x)| \leq \varepsilon$, then return $w_v K((v_l - v_s)/2, x)$ to the parent. Else, visit each child and return their sum to the parent. This approach may improve the query evaluation time in practice, especially when $|Q|$ is large.

5. TWO DIMENSIONS

In $\mathbb{R}^2$ we first describe baseline methods from the literature or based on simple extensions to existing methods. We then introduce our new methods. The first uses a randomized technique based on matchings, and the second is deterministic and based on space-filling curves.

5.1 Baseline Methods

Random sampling (RS). The first baseline (labeled RS) is to simply random sample a set Q from P. The same bound of $O((1/\varepsilon^2)\log(1/\delta))$ on the sample size from one dimension still holds in 2-dimensions, although the constant in the big-Oh is likely larger by a factor of about 2.

Improved fast Gauss transform (IFGT). A class of methods is based on *fast multi-pole methods* [4]. In practice in 2-dimensions these are implemented as quad trees which calculate the distance to roots of subtrees instead of all pairs when the distance becomes far enough. The Improved Fast Gauss Transform (IFGT) [14,36,49,50], is the state-of-the-art for fast construction and evaluation of approximate kernel density estimates (although only with Gaussian kernels). It improves multi-pole approaches by first building a k-center clustering over the data set P, and then just retaining a Hermite expansion of the kernel density estimates for the points associated with each k-centers. But the IFGT method is based on heuristics and does not offer any formal theoretical guarantees on the approximation-size trade-off. As a result, it involves a number of parameters that cannot be easily and intuitively set in order to derive a desired level of efficiency and accuracy tradeoffs.

Kernel herding (KH). Yet another possible approach is to explore the *reproducing kernel Hilbert space* (RKHS). As discussed in Section 3, these RKHS approximations are typically constructed through some form of random sampling [41,48], but one technique known as "kernel herding" [7] uses a greedy approach and requires significantly smaller size in theory, however it bounds only ℓ_2 error as opposed to the sampling techniques which bound a stronger ℓ_∞ error.

In particular, the state-of-the-art kernel herding technique from [7] is a greedy method. It adds at each step the single point $p \in P \setminus Q$ to Q which most decreases $\|\text{KDE}_Q - \text{KDE}_P\|_2$. This is possible to calculate efficiently through an approximate representation of KDE_Q and KDE_P in the RKHS. However, this still takes $O(|Q|n)$ time to construct $|Q|$ since at each step each point in $P \setminus Q$ needs to be evaluated to determine how much it will decrease the ℓ_2 error.

5.2 Randomized MergeReduce Algorithm

An interesting theoretical result built on discrepancy [5, 30] theory was recently proposed in [33] for constructing a small set Q so KDE_Q approximates KDE_P. It extends a classic method for creating ε-approximations of Chazelle and Matousek [6] (see also [5,30]), to ε-approximate kernel density estimates. These results are mostly of theoretical interest, the straightforward adaption is highly inefficient. Next we explain and overcome these inefficiencies.

5.2.1 The MergeReduce framework

Our algorithm leverages the framework of Chazelle and Matousek [6] and its generalizations. We first describe our overall framework, and then elaborate the most critical *reduce step* in further details. Roughly speaking, our algorithm repeatedly runs a merge-then-reduce step. Hence, we denote this framework as the *MergeReduce* algorithm.

Suppose the desired size of the compressed set Q is $|Q| = k$. The framework proceeds in three phases: an initialization

phase, the combination phase, and the optional clean-up phase. The first phase is implicit, and the last phase is of theoretical interest only.

In the initialization phase we arbitrarily decompose P into disjoint sets of size k; call these $P_1, P_2, \ldots, P_{n/k}$. Since this is arbitrary, we can group data that is stored together into the same partition. This works well for distributed data or streaming data where consecutively encountered data are put in the same partition.

The combination phase proceeds in $\log(n/k)$ rounds. In each round, of the remaining sets of size k, it *arbitrarily* pairs them together, which we dub the *merge step*. For each pair say P_i and P_j, it runs *a reduce step* on the union of $2k$ points to create a single set of size k. At a high level, the *reduce step* has two parts. The first part is what we call a *matching operation*. It produces k pairs of points in a certain fashion over the $2k$ input points. The second part is trivial: it randomly selects one point from each pair to produce the final output of size k. The *merging operation* is the most critical part in a reduce step, and we will elaborate after presenting the overall MergeReduce framework.

That said, after i rounds of *merge and reduce*, there are $(n/k)/2^i$ sets of size k. This continues until there is one set remaining. Note that again the fact that we can pair sets (P_i and P_j) arbitrarily in a *merge step* is extremely convenient in a distributed or streaming setting.

The clean-up phase is not needed if the combination phase is run as above; see Section 5.2.4 for remaining details.

Importantly, we also note that this entire MergeReduce framework can be preceded by randomly sampling $O(1/\varepsilon^2)$ points from P which are then treated as the input. Then only $\log(1/\varepsilon)$ merge-reduce rounds will be needed.

The min-cost matching. The key part of this framework is to construct a *matching* in a set of points P. Suppose $|P| = n$, a *matching* consists of $\frac{n}{2}$ pairs of points, and every point in P belongs to exactly one pair in a *matching*.

The recent result from [33] implies that one can use the min-cost matching to derive an ε-approximate kernel density estimate. Note that a min-cost matching is a matching so that the sum of distances between matched points is minimized. Unfortunately, by using a min-cost matching, the algorithm in [33] is impractical. Here is why.

It is well known that a min-cost matching over n points can be done in $O(n^3)$ time using Edmonds' Blossom algorithm [11]. This quite complicated algorithm involves non-regular recursion, and it is clearly not scalable for large data sets. We use the state-of-the-art implementation [25] as a baseline for the matching operation and label it as Blossom-MR (MergeReduce with Blossom min-cost matching). This implementation of the Blossom algorithm requires first calculating $K(p, p')$ for all $O(n^2)$ pairs $p, p' \in P$ as input, which is part of the overall cost (which is $O(n^3)$).

There have been theoretical improvements [45] to Edmonds' algorithm for points in $\mathbb{R}^2$. These algorithms are considerably more complicated than that of Edmonds and no known efficient implementation exists; most likely the improvements are theoretical in nature only.

We have the following results concerning the Blossom-MR algorithm, and the Blossom-MR+RS algorithm that first randomly samples $O(1/\varepsilon^2)$ points.

Theorem 2 *For a point set $P \subset \mathbb{R}^2$ with n points, we can construct Q giving an ε-approximate KDE (with constant probability) in*

- $O(\frac{n}{\varepsilon^2}\log^2 n \log\frac{1}{\varepsilon})$ *time and* $|Q| = O(\frac{1}{\varepsilon}\log n\sqrt{\log\frac{1}{\varepsilon}})$ *using Blossom-MR, and*
- $O(n + \frac{1}{\varepsilon^4}\log^3\frac{1}{\varepsilon})$ *time and* $|Q| = O(\frac{1}{\varepsilon}\log^{1.5}\frac{1}{\varepsilon})$ *using Blossom-MR+RS.*

Lastly, the following greedy algorithm provides a two-approximation to the cost of the min-cost matching [10]. It finds the closest pair of points in P, matches them, removes them from P, and repeats. This algorithm can be implemented in $O(n^2 \log n)$ time as follows. Calculate all $O(n^2)$ pairwise distances and place them in a priority queue. Repeatedly remove the smallest pair from the queue (in $O(\log n)$ time). If both points are still in P, match them and mark them as no longer in P. We refer to the merge-reduce framework with this matching algorithm as the Greedy-MR method. Greedy-MR does improve the running time over Blossom-MR, however, it is still quite expensive and not scalable for large data. Furthermore, the result produced by Greedy-MR is not known to provide any approximation guarantees on the kernel density estimate.

5.2.2 More Efficient Reduce Step

The Blossom-MR algorithm and its heuristic variant Greedy-MR are too expensive to be useful for large data. Thus, we design a much more efficient *matching operation*, while still ensuring an ε-approximate kernel density estimate.

For any matching M, we produce an *edge map* E_M of that matching M as $E_M = \{e(p,q) \mid (p,q) \in M\}$ where $e(p,q)$ is an undirected edge connecting p and q. Given a disk B, for $e(p,q) \in E_M$ define $e(p,q) \cap B$ as follows.

- If both p and q are not covered by B, $e(p,q) \cap B = \emptyset$.
- If both p and q are covered by B, $e(p,q) \cap B = e(p,q)$.
- If either p or q is covered by B but not both. Suppose p is covered by B, and $e(p,q)$ intersects the boundary of B at a point s, $e(p,q) \cap B = e(p,s)$.

Then, we define $E_M \cap B$ as:

$$E_M \cap B = \{e(p,q) \cap B \mid e(p,q) \in E_m\}. \tag{5}$$

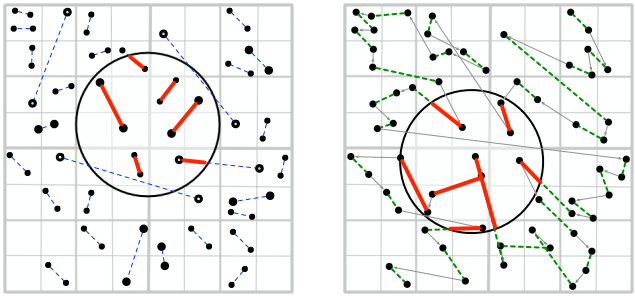


Figure 2: Intersection between a matching and a disk, solid red lines are included in $E_M \cap B$. Left shows Grid matching and right shows Z-order with every other edge in a matching.

An example of $E_M \cap B$ is shown in Figure 2. Essentially, we only want it to consider edges with at least one endpoint within B, and only the subset of the edge that is within B. We observe that in order to produce an ε-approximate kernel density estimate, the property the matching requires is in regards to any unit disk B. Specifically, we want

$$C_{M,B} = \sum_{e(p,q) \in E_M \cap B} \|p - q\|^2 \tag{6}$$

to be small. Let $C_M = \max_B C_{M,B}$.

The result in [33] says if M is the min-cost matching (minimizes $\sum_{(p,q)\in M} \|p-q\|$), then $C_M = O(1)$. But calculating the min-cost matching, both exactly and approximately, is expensive as we have shown in Section 5.2.1.

The grid matching. Here we present a novel solution which does have a bound on C_M and is efficient, including at very large scales. We progress in rounds until all points are matched. Starting with $i = 0$, in round i we consider a grid G_i on $\mathbb{R}^2$ where each grid cell has edge length $l_{\varepsilon,i} = \sqrt{2}\sigma\varepsilon 2^{i-2}$. Define cell $g_{r,c} \in G_i$ as $[rl_{\varepsilon,i}, (r+1)l_{\varepsilon,i}] \times [cl_{\varepsilon,i}, (c+1)l_{\varepsilon,i}]$ for integers r, c. Inside of each cell, match points arbitrarily. Only the unmatched points (one from any cell with an odd number of points) survive to the next round. Each cell in G_{i+1} is the union of 4 cells from G_i, thus in rounds $i > 0$ it can have at most 4 points. We refer to this matching algorithm as Grid; see example in Figure 2. For simpler analysis, we assume $\sigma > \varepsilon^c$ for some constant $c \geq 1$; typically $\sigma \gg \varepsilon$ and $\varepsilon < 1$, so this assumption is very mild. Alternatively, if we do not know σ ahead of time, we can recursively divide points that fall in the same grid cell into smaller and smaller levels until at most two are left in each cell (like a quad tree), and then move back up the recursion stack only with unmatched points; a careful implementation can be done in $O(n \log n)$ time [18]. Let P_0 be unmatched points after round 0, let $P' = P \setminus P_0$, and Q the final output.

Lemma 3 *Let $M'(P')$ be the matching on P' in Grid. For each edge $(p, q) \in M'(P')$ let $\hat{P}$ be where (w.l.o.g.) q is moved to location p. Then $\hat{P}$ is a $\varepsilon\sigma/2$-perturbation of P'.*

Proof. Since all points matched in round 0 are in a grid cell of size at most $\varepsilon\sigma\sqrt{2}/4$, the point q in any edge is moved at most $\sqrt{2} \cdot \varepsilon\sigma\sqrt{2}/4 = \varepsilon\sigma/2$. □

Lemma 4 *Let $M_0(P_0)$ be the matching by Grid on P_0. Then $C_{M_0} = O(\log(1/\varepsilon))$ and Grid runs in $O(n \log \frac{1}{\varepsilon})$ time.*

Proof. In each round, there are at most 2 matched pairs per grid cell. Each such pair has edge length at most $\sqrt{2}l_{\varepsilon,i} = \sigma\varepsilon 2^{i-1}$, and there are at most $(1/\sigma\varepsilon 2^{i-5/2})^2$ grid cells that intersect any unit ball. Thus the total squared-length of all matchings in round i is at most $((1/\sigma\varepsilon 2^{i-5/2})^2 \cdot (\sigma\varepsilon 2^{i-1})^2 = 2\sqrt{2}$. After $\log(1/\sigma\varepsilon) + 1$ rounds the total length of all matchings is at most $2\sqrt{2}(\log(1/\sigma\varepsilon) + 1)$, and in any ball, there are at most 4 remaining unmatched points. The last 4 points can each account for at most a squared-length of 4 within a unit ball B, so the total weight of edges in any unit ball B is at most $C_M \leq 2\sqrt{2}\log(1/\sigma\varepsilon) + 19 = O(\log(1/\varepsilon))$. Each round takes $O(n)$ time, and we can match points arbitrarily in $O(n)$ time after the $\log(1/\sigma\varepsilon) + 1$ rounds. □

We observe in most common scenarios C_M is close to 1.

With the Grid matching algorithm, we can instantiate the MergeReduce framework to get a Grid-MR algorithm, or if we first sample a Grid-MR+RS algorithm.

Theorem 3 *For a point set $P \subset \mathbb{R}^2$ with n points, we can construct Q giving an ε-approximate KDE (with constant probability) in*

- *$O(n \log \frac{1}{\varepsilon})$ time and $|Q| = O(\frac{1}{\varepsilon} \log n \log^{1.5} \frac{1}{\varepsilon})$ using Grid-MR, and*
- *$O(n + \frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon})$ time and $|Q| = O(\frac{1}{\varepsilon} \log^{2.5} \frac{1}{\varepsilon})$ using Grid-MR+RS.*

Since Grid takes only $O(n \log \frac{1}{\varepsilon})$ time, the benefit of the initialization phase to split the data set into n/k pieces does not out-weigh its overhead in a centralized setting. In particular, we just run Grid once on all n_i points remaining in round i. This does not affect the runtime or error bounds.

Compared to the Blossom-MR and Greedy-MR algorithms, Grid-MR produces an ε-approximate kernel density estimate with about the same size, but with much cheaper construction cost. Grid-MR's running time only linearly depends on n, making it an ideal candidate to scale out to massive data.

5.2.3 Streaming and Distributed MergeReduce

Since the reduce step (the key computational component of the MergeReduce framework) is only run on select subsets, this allows the full framework to generalize to distributed and streaming settings.

Streaming variant. The streaming algorithm follows techniques for approximate range counting [3, 43]. Consecutive points are assigned to the same partitions P_i, and we pair and reduce partitions whenever there are two that represent the same number of points (one that has been merged i times represents $k2^i$ points). This means we only need to keep $\log \frac{n}{k}$ partitions, and thus only $O(k \log \frac{n}{k})$ points in memory at any given time. The dependence on n can be completely removed by first randomly sampling.

The final structure of the streaming algorithm has weighted points, where if a point is in a partition that has been reduced i times, its weight is 2^i. These weights can be removed to create just $5|Q|$ points instead of $|Q| \log \frac{|P|}{|Q|}$ by running the in memory matching algorithm on weighted points [29].

In particular, we can modify a matching algorithm to work with weighted points, specifically consider the Grid algorithm. In the 0th phase of Grid, a point with weight 2^i represents 2^i points that all fall in the same cell, and can be matched with themselves (this can be done by ignoring this point until the ith phase when its weight is 1).

Distributed variant. This framework is efficient on distributed data. Use the streaming algorithm on each distributed chunk of data, and then pass the entire output of the streaming algorithm to a central node, which can merge and reduce the union. The error caused by reducing on the central node is already accounted for in the analysis. Again, the dependence on $|P|$ can be removed by first sampling.

5.2.4 Other Extension

An alternative version of the combination phase is possible for the MergeReduce algorithm. Specifically, it considers some reduce step that takes time $O(n^\beta)$ on a set of size n, and instead sets $k = 4(\beta + 2)|Q|$ (where $|Q|$ is the desired size of the final output), and on every $(\beta+3)$th round, pairs sets but does not reduce them. Then the clean-up phase is used to reduce the single remaining set repeatedly until it is of size $|Q|$. When β is a constant, this saves a $O(\log n)$ from the size of the output Q (or $O(\log \frac{1}{\varepsilon})$ if we sampled first) [6].

More specifically, the output of this MergeReduce variant is then a set of size $|Q| = O(C_M \frac{1}{\varepsilon} \log^{0.5} \frac{1}{\varepsilon})$ for a reduce step that uses a matching algorithm which produces an output with cost C_M. If we use Grid, then Grid-MR produces an output of size $|Q| = O(\frac{1}{\varepsilon} \log^{1.5} \frac{1}{\varepsilon})$. And Blossom-MR outputs Q of size $|Q| = O(\frac{1}{\varepsilon} \log^{0.5} \frac{1}{\varepsilon})$ in $O(\frac{n}{\varepsilon^2} \log \frac{1}{\varepsilon})$ time.

But in practice, this variant is more complicated to implement and usually the overhead out-weighs its benefit. Hence we do not use this variant in this paper.

5.3 Deterministic Z-Order Selection

Inspired by one-dimensional sort-section (SS) and randomized two-dimensional Grid-MR algorithm, we propose a new deterministic two-dimensional technique based on space filling curves. A space filling curve puts a single order on two- (or higher-) dimensional points that preserves spatial locality. They have great uses in databases for approximate high-dimensional nearest-neighbor queries and range queries. The single order can be used for a (one-dimensional) B+ tree, which provides extremely efficient queries even on massive datasets that do not fit in memory.

In particular, the Z-order curve is a specific type of space filling curve that can be interpreted as implicitly ordering points based on the traversal order of a quad tree. That is if all of the points are in $[0,1]^2$, then the top level of the quad tree has 4 children over the domains $c_1 = [0, \frac{1}{2}] \times [0, \frac{1}{2}]$, $c_2 = [\frac{1}{2}, 1] \times [0, \frac{1}{2}]$, $c_3 = [0, \frac{1}{2}] \times [\frac{1}{2}, 1]$, and $c_4 = [\frac{1}{2}, 1] \times [\frac{1}{2}, 1]$. And each child's four children itself divide symmetrically as such. Then the Z-order curve visits all points in the child c_1, then all points in c_2, then all points in c_3, and all points in c_4 (in the shape of a backwards 'Z'); and all points within each child are also visited in such a Z-shaped order. See an example in Figure 2. Thus given a domain containing all points, this defines a complete order on them, and the order preserves spatial locality as well as a quad tree does.

The levels of the Z-order curve (and associated quad tree) are reminiscent of the grids used in the matching technique Grid. This insight leads to the following algorithm.

Compute the Z-order of all points, and of every two points of rank $2i$ and $2i+1$, discard one at random; repeat this discarding of half the points until the remaining set is sufficiently small. This approach tends to match points in the same grid cell, as with Grid, but is also algorithmically wasteful since the Z-order does not change between rounds.

Thus we can improve the algorithm by using insights from SS. In particular, we just run SS on the Z-order of points. So to collect $k = |Q|$ points, let the ith point retained $q_i \in Q$ be the point in rank order $\lceil (i - \frac{1}{2})\frac{|P|}{k} \rceil$. This selects one point from each set of $\frac{|P|}{k}$ points in the Z-order.

In fact, by setting $k = O(\frac{1}{\varepsilon} \log n \log^{1.5} \frac{1}{\varepsilon})$, if we randomly select one point from each Z-order rank range $[(i-1)\frac{|P|}{k}, i\frac{|P|}{k}]$ (call this algorithm Zrandom), then the resulting set Q has about the same guarantees as the Grid-MR algorithm, including Zrandom+RS which preprocesses by random sampling.

Theorem 4 *For a point set $P \subset \mathbb{R}^2$ with n points, we can construct Q giving an ε-approximate KDE (with constant probability) in*

- *$O(n \log n)$ time and $|Q| = O(\frac{1}{\varepsilon} \log n \log^{1.5} \frac{1}{\varepsilon})$ using Zrandom, and*
- *$O(n + \frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon})$ time and $|Q| = O(\frac{1}{\varepsilon} \log^{2.5} \frac{1}{\varepsilon})$ using Zrandom+RS.*

Proof. We prove this result by imagining that Zrandom does something more complicated than it actually does in order to relate it to Grid-MR. That is, we pretend that instead of just selecting a single points at random from each range $[(i-1)|P|/|Q|, i|P|/|Q|]$ in the Z-order rank, we proceed in a series of $\log(|P|/|Q|)$ rounds, and in each round the set P is reduced in size by half. Specifically, in each round, out of every two consecutive points in the Z-order (rank $2i$ and $2i+1$) we retain one at random. Since the Z-order is consistent across rounds, this results in a random point in the rank interval $[(i-1)|P|/|Q|, i|P|/|Q|]$ as desired.

Now if we consider the levels of the implicit quad tree defining the Z-order, this is equivalent to the grid used in Grid-MR. In each round, there are at most 3 edges within grid cell at level j, but that are not entirely in levels smaller than j. Since we still only care about $O(\log(1/\varepsilon))$ levels of the grid, the squared distance of these edges in that cell level account for at most $O(1)$ inside a unit square. Thus C_M is still at most $O(\log(1/\varepsilon))$. The remainder of the proof is the same as in Theorem 3. □

However, we find the implementation of the following deterministic algorithm to be more effective; but as the randomness is necessary for the proof, we do not provide bounds.

The construction of the Z-order can be done efficiently using its interpretation as bit-interleaving. Specifically, the *z-value* of a point is calculated by interleaving bits from the most significant bit to the least significant bit in the binary representation of a point's coordinates. For example, a two-dimensional point $(3, 5)$ expressed in its binary representation is $(011, 101)$. Its *z-value* is then $(011011)_2 = 27$.

Then we do not need to completely sort all points by z-value in order to select the proper k points, we can just approximately do this so that we have one point selected from each set of $\frac{|P|}{k}$ points in the sorted order. This can be done using an ε-approximate quantiles algorithm that is accurate up to $\varepsilon = 1/k$. This guarantees the existence in the quantile structure and its identification of at least one point within every consecutive set of $\frac{|P|}{k}$ points. We can just return this point for each range $[(i-1)\frac{|P|}{k}, i\frac{|P|}{k}]$ of ranks. We refer to this method as Zorder. Note that, following the discussion for SS in Section 4, we can use any of the existing efficient, large-scale ε-approximate quantiles algorithms in either the centralized or the distributed setting.

5.4 Efficient Query Evaluation

We can do efficient evaluations in $\mathbb{R}^2$ similar to BF and MP in $\mathbb{R}^1$, as discussed in Section 4.1. In fact, BF is exactly the same. MP uses a quad tree instead of a binary tree. It stores a bounding box at each node instead of an interval, but uses the same test to see if the difference between $K(x, \cdot)$ is within ε on the furthest and closest point in the bounding box to decide if it needs to recur.

5.5 Parallel and Distributed Implementation

As with one-dimensional methods, our two-dimensional methods can be efficiently implemented in distributed and streaming settings. Any algorithm using the MergeReduce framework can run in distributed and parallel fashion. As a result, Grid-MR, Zrandom, and Zorder are very efficient in any distributed and parallel environments, and they extend especially well for the popular MapReduce framework where data in each split is processed in a streaming fashion.

Among the baseline methods, RS can easily be implemented in any distributed and parallel environments. It takes some effort to make IFGT run in distributed and parallel fashion, but it is possible; we omit the details. Lastly, the KH can be approximated (without any bounds) by running its greedy step in each local piece of data independently, and then merging the results from local nodes.

5.6 Higher Dimensions

All two-dimensional algorithms described can be extended to higher dimensions. In particular, KH, RS, Blossom-MR,

Greedy-MR extend with no change, while Grid-MR and Zorder-MR extend in the obvious way of using a d-dimension grid or space-filling curve. In $\mathbb{R}^d$ the theoretical size bounds for RS increases to $O(\frac{1}{\varepsilon^2}(d+\log\frac{1}{\delta}))$ [24]; Grid-MR, and Zrandom-MR increases to $O(d^{d/2}/\varepsilon^{2-\frac{4}{d+2}}\log^{1+\frac{d}{d+2}}\frac{1}{\varepsilon}\log n)$ (and a $\log\frac{1}{\varepsilon}$ factor less for Blossom-MR). The increase in the second set is due to only being able to bound

$$C_M^d = \max_B \sum_{e(p,q)\in E_M\cap B} \|p-q\|^d$$

instead of equation (6) since the number of grid cells intersected by a unit ball now grows exponentially in d, and thus we need to balance that growth with the dth power of the edge lengths. The stated bounds, then results from [33] with an extra $\log n$ factor (which can be turned into a $\log\frac{1}{\varepsilon}$ by first random sampling) because we do not use the impractical process described in Section 5.2.4. In no case does the MergeReduce framework need to be altered, and so the construction times only increase by a factor d.

6. EXPERIMENTS

We test all methods in both the single-thread, centralized environment and the distributed and parallel setting. In the centralized case, we implemented all methods in C++ and obtained the latest implementation of the IFGT method from the authors in [36, 49, 50]. We then used MapReduce as the distributed and parallel programming framework. In particular, we implemented and tested all methods in a Hadoop cluster with 17 machines. The centralized experiments were executed over a Linux machine running a single Intel i7 cpu at 3.20GHz. It has 6GB main memory and an 1TB hard disk. The distributed and parallel experiments were executed over a cluster of 17 machines running Hadoop 1.0.3. One of the 17 machines has an Intel Xeon(R) E5649 cpu at 2.53 GHz, 100 GB of main memory, and a 2TB hard disk. It is configured as both the master node and the name node of the cluster. The other 16 machines in the Hadoop cluster (the slave nodes) share the same configuration as the machine we used for the centralized experiments. One TaskTracker and DataNode daemon run on each slave. A single NameNode and JobTracker run on the master. The default HDFS (Hadoop distributed file system) chunk size is 64MB.

Data sets. We executed our experiments over two large real datasets. In two dimensions, we used the OpenStreet data from the OpenStreetMap project. Each dataset represents the points of interest on the road network for a US state. The entire dataset has the road networks for 50 states, containing more than 160 million records in 6.6GB. For our experiments we focus on only the 48 contiguous states, excluding Hawaii and Alaska. Each record is a 2-dimensional coordinate, represented as 2 4-byte floating points. We denote this data as the *US* dataset.

In one dimension, we employed the second real dataset, *Meme*, which was obtained from the Memetracker project. It tracks popular quotes and phrases which appear from various sources on the internet. The goal is to analyze how different quotes and phrases compete for coverage every day and how some quickly fade out of use while others persist for long periods of time. A record has 4 attributes, the URL of the website containing the memes, the time Memetracker observed the memes, a list of the observed memes, and links accessible from the website. We preprocess the *Meme* dataset, and convert each record to have an 8-byte double to represent the *time* of a single observed meme and the URL of the website which published the meme, e.g. if an original record contained a list of 4 memes published at a given time at a website, 4 records would be produced in the new dataset. We view these records as a set of timestamps in 1d, reflecting the distribution of the Meme's publication time. After this preprocessing step, the *Meme* dataset contains more than 300 million records in 10.3GB.

In both 1d and 2d, whenever needed, we randomly sample records from the full *US* or the full *Meme* dataset to obtain a dataset of smaller size. Figure 1 in Section 1 visualizes the kernel density estimates built by our MergeReduce algorithms in 1d and 2d, over the full *Meme* and *US* datasets respectively (but only very few data points were plotted, to ensure that the figures are legible).

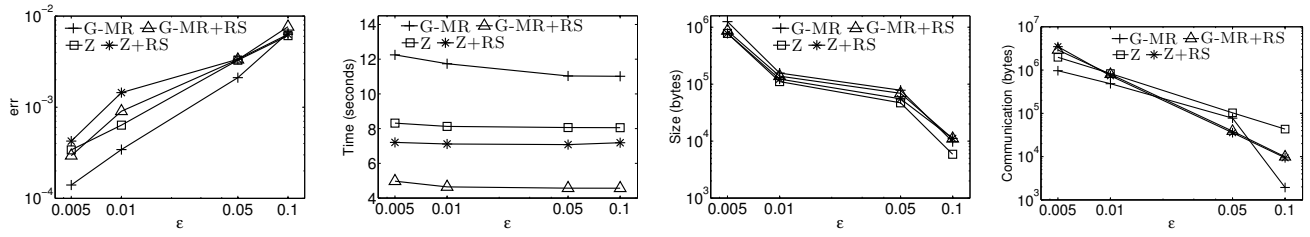
General setup. In all experiments, unless otherwise specified, we vary the values of one parameter of interest, while keeping the other important parameters at their default values. Also by default, we randomly generate 5,000 test points to evaluate the accuracy of an approximate kernel density estimate. Among these 5,000 points, 4,000 were randomly selected from the data set P, and the other 1,000 were randomly selected from the domain space $\mathcal{M}$ of P (but not from P itself). We use *err* to denote the *observed* ℓ_∞ *error* from an approximate kernel density estimate Q, which is computed from the evaluations of these 5,000 test points in KDE_Q and KDE_P respectively. We try to compare different methods by setting a proper ε value (the desired error in theory) for each of them so that they achieve the same observed error. All experiments report the *average of 10 random trials*.

6.1 Two Dimensions: Centralized

Default setup. Our default data set is a *US* data set with 10 million records. The default failure probability δ for the random sampling method (RS) is set to 0.001. To save space in the legend in all figures, we used G-MR and Z to denote the Grid-MR and Zorder methods respectively, and method+RS to denote the version of running method over a random sample of P (of size $O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta})$), instead of running method over P directly. The default σ value in any kernel density estimate is 200, on a domain of roughly 50,000 × 50,000.

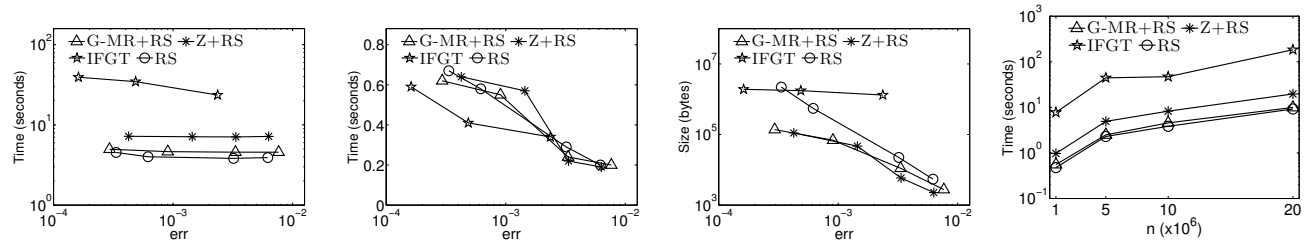
Our method+RS. We first study the effect of running our methods over a random sample of P, when compared against the results from running them directly over P. Figure 3 shows the results when we vary the value of the common input parameter for all of them, ε. Not surprisingly, as shown in Figure 3(a), the observed errors in all methods are smaller than the desired error ε. All methods produce smaller observed errors when smaller ε values were used. Furthermore, under the same ε value, G-MR+RS and Z+RS produce higher errors than their respective counterpart does, namely, G-MR and Z. However, the difference is fairly small. In fact, when *err* is about 10^{-2}, there are almost no difference between G-MR+RS (Z+RS) and G-MR (Z).

More importantly however, running a method over a random sample of P saves valuable construction time as shown in Figure 3(b). Figure 3(c) indicates the sizes of the final approximate kernel density estimates constructed by different methods are almost the same. This is because that whether running a method over P or a random sample of P, the final size of the kernel density estimate Q is largely determined by ε only. This also means that the query time of these



(a) *err* vs. ε. (b) Construction time. (c) Size. (d) Communication.

Figure 3: Effect of guaranteed error ε on the centralized G-MR, G-MR+RS, Z, Z+RS.



(a) Construction time vs *err*. (b) Query Time. (c) Size. (d) Construction time vs n.

Figure 5: Effect of measured ℓ_∞ error *err* and N on centralized IFGT, RS, G-MR+RS, Z+RS.

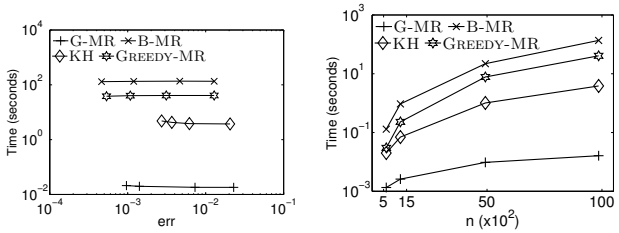
methods is almost the same, since the query time depends on only $|Q|$. Hence, we have omitted this figure.

Finally, we also investigate the impact to the communication cost if we run these methods in a cluster. Figure 3(d) shows that this impact is not obvious. Since G-MR and Z are very effective in saving communication cost already, collecting a random sample first does not lead to communication savings. In contrast, doing so might even hurt their communication cost (if the sample size is larger than what they have to communicate in our MergeReduce framework). Nevertheless, all methods are very efficient in terms of the total bytes sent: a few megabytes in the worst case when $\varepsilon = 0.005$ over a cluster for 10 million records in P.

In conclusion, these results show that in practice one can use our method+RS to achieve the best balance in construction time and accuracy for typically required observed errors.

Our method vs. baseline methods. We first investigate the construction cost of different alternatives in instantiating our MergeReduce framework, with different algorithm for the matching operation. We also include Kernel Herding (KH) as introduced in Section 5.1. In order to let these baseline methods complete in a reasonable amount of time, we used smaller data sets here. From the analysis in Section 5.2, Blossom-MR (denoted as B-MR, representing the theoretical algorithm [33]) with a $O(nk^2)$ cost for output size k and Greedy-MR with a $O(nk \log n)$ cost are much more expensive than G-MR. Similarly, KH with a $O(nk)$ cost is also much more expensive than G-MR which runs in roughly $O(n \log k)$ time. This is even more pronounced in Figures 4(a) and 4(b), showing the construction time of different methods when varying the observed error *err* and the size of the data sets respectively. G-MR is several orders of magnitude faster and more scalable than the other methods.

So the only competing baseline methods we need to consider further are the RS and IFGT methods. We compare these methods against our methods, G-MR+RS and Z+RS in Figure 5, using the default 10 million *US* data set. Figure 5(a) indicates that, to achieve the same observed error, RS is the most efficient method in terms of the construction time. However, our methods G-MR+RS and Z+RS are almost as efficient. In contrast, IFGT is almost one order of magnitude slower. In terms of the query time, Figure 5(b) shows that all methods achieve a similar query time given the same observed error, though IFGT does slightly better for very small *err* values. Note that we used the *multipole* (MP) query evaluation method for the kernel density estimates built from RS, G-MR+RS, and Z+RS. On the other hand, Figure 5(c) shows that the kernel density estimates built from both IFGT and RS have much larger size than that produced in our methods G-MR+RS and Z+RS, by 2 to 3, and 1 to 2 orders of magnitude respectively. Finally, Figure 5(d) indicates that all methods have very good scalability in their construction time when the data set size increases from 1 million to 20 million, but G-MR+RS, Z+RS, and RS are clearly much more efficient than IFGT.



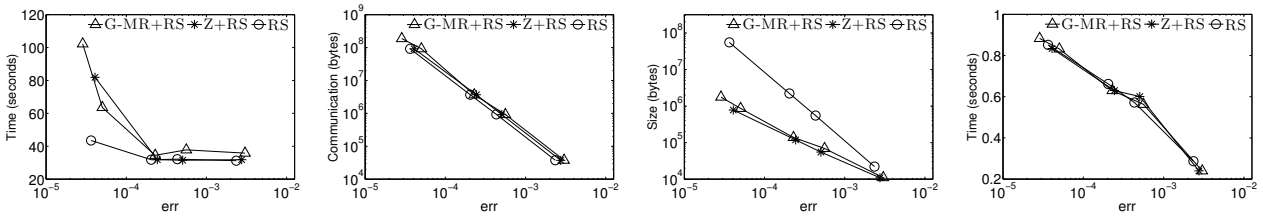
(a) Build time vs. *err*. (b) Build time vs. n.

Figure 4: Centralized G-MR, B-MR, Greedy-MR, KH.

In conclusion, our methods are almost as efficient as RS in terms of building a kernel density estimate, and they are much more efficient than IFGT. Our methods also share similar query time as IFGT and RS, while building much smaller kernel density estimates (in size) than both IFGT and RS.

6.2 Two Dimensions: Parallel and Distributed

Default setup. In this case, we change the default data set to a *US* data set with 50 million records, while keeping the other settings the same as those in Section 6.1. Furthermore, since IFGT is much slower in building a kernel density estimate (even more so for larger data sets), and it is also a heuristic without theoretical guarantees (in contrast to the other 3 methods), in the distributed and parallel setting, we focus on comparing our methods against the RS



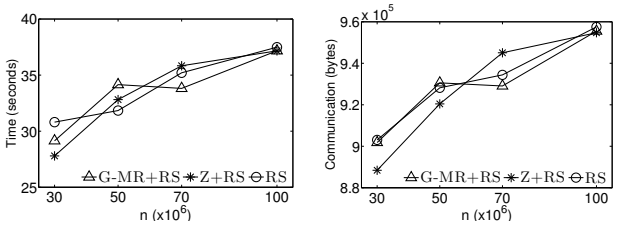
(a) Construction time. (b) Communication. (c) Size. (d) Query time.

Figure 6: Effect of measured ℓ_∞ error *err* on distributed and parallel G-MR+RS, Z+RS, RS.

method. Moreover, IFGT works only for the Gaussian kernel and needs to be provided the bandwidth σ ahead of time, whereas our methods and RS do not. For space, we omit experiments showing varying σ has mild effects on our algorithms and RS, but can lead to strange effects in IFGT.

Our methods vs. RS. We compare the performance of our methods, G-MR+RS and Z+RS, against RS on the aforementioned Hadoop cluster. Figure 6 reports the results when we vary the observed error *err* for different methods (by adjusting their ε values properly so they output roughly the same *err*). Figure 6(a) shows that RS is the most efficient method to construct an approximate kernel density estimate for small *err* values, but G-MR+RS and Z+RS become almost as efficient as RS once *err* is no smaller than 10^{-4} which is sufficient for many practical applications. In those cases, all three methods take less than 40 seconds to construct an approximate KDE for 50 million records. All methods are highly communication-efficient, as shown in Figure 6(b). There are almost no difference among the 3 methods: they communicate only a few MBs over the cluster to achieve an *err* of 10^{-4}, and tens or hundreds of KBs for *err* between 10^{-2} and 10^{-3}. In terms of the size of the approximate KDE_Q, not surprisingly, RS is always the largest. By our analysis in Sections 5.2 and Sections 5.3, $|Q|$ is $O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta})$ for RS, and only $O(\frac{1}{\varepsilon}\log^{2.5}\frac{1}{\varepsilon})$ for both G-MR+RS and Z+RS. This is clearly reflected in Figure 6(c), where $|Q|$ is 1-2 orders of magnitude larger from RS than from G-MR+RS and Z+RS. Finally, we investigate the query time using Q in Figure 6(d). In general, the query time should be linear to $|Q|$. But in practice, since we have used the fast query evaluation technique, the *multipole* (MP) method as shown in Section 5.4, all three methods have similar query time.

We thus conclude that our methods, G-MR+RS and Z+RS perform better than RS. More importantly, they also have much better theoretical guarantees (in order to achieve to same desired error ε in theory), which is critical in practice since users typically use a method by setting an ε value, and for the same ε value, our methods will outperform RS by orders of magnitude ($O(\frac{1}{\varepsilon}\log^{2.5}\frac{1}{\varepsilon})$ vs. $O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta})$). Nevertheless, to be as fair as possible, we experimentally compared methods by first running a few trials to set a proper ε value so each method has roughly the same observed error *err*.



(a) Time. (b) Communication.

Figure 7: Effect of n on G-MR+RS, Z+RS, RS.

We also study the scalability of these methods in Figure 7 by varying the size of the data from 30 million to 100 million records. Not surprisingly, all three methods are extremely scalable in both their construction time (almost linear to n) and communication cost (almost constant to n). Communication only increases by 0.8×10^5 bytes when the total communication is about $9 - 10 \times 10^5$ bytes; such increase is due to the overhead in Hadoop to handle more splits (as n increases), rather than the behavior of our algorithms which is independent from n in communication cost.

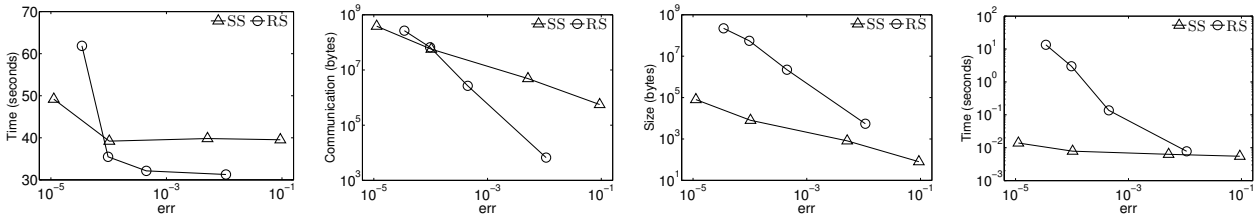
6.3 Two Dimensions: Is RS Good Enough?

To show our advantages over random sampling, we provide more centralized experiments here for higher precision. The trends in the parallel and distributed setting will be similar.

Getting higher precision results from a kernel density estimate can be very desirable. The error of kernel density estimates over samples is with respect to $\text{KDE}_P(x) = \frac{1}{|P|}\sum_{p\in P} K(p,x)$, but for large spatial data sets, often only a small fraction of points have non-negligible effect on a query x, so dividing by $|P|$ can make $\text{KDE}_P(x)$ quite small. Here it can be of critical importance to have very small error. Another use case of kernel density estimates is in n-body simulations in statistical physics [1], where high precision is required to determine the force vector at each step. Furthermore, note that a user typically uses these methods with a desirable error as the input, which is set as the input error parameter, the ε value; even though the observed error *err* on a data set may be smaller than ε. In that case, all of our methods have (roughly) a $O(\frac{1}{\varepsilon}/\log\frac{1}{\varepsilon})$ factor improvement in the KDE size, which is a critical improvement especially when ε needs to be small (for high precision applications).

We observe experiments in Figure 8 which compares G-MR+RS and Z+RS with RS in terms of construction time and size of the samples. (Note that figures for query time were omitted for the interest of space; but not surprisingly, they are roughly linear to the KDE size). We plot the figures based on input error parameter ε and observed error *err*. For the plot with respect to ε (Figure 8(a), 8(c)), when ε becomes smaller than $5 * 10^{-4}$, the construction time and size for RS remain constant as the sample size needed for RS becomes larger than the size of the original dataset. Since we then don't need to sample, the construction time and observed error for RS are 0. For small ε values, G-MR+RS and Z+RS are clever enough to test if random sampling is beneficial, and if not the random sampling step is bypassed.

For the higher precision observed error, the results (Figures 8(b), 8(d)) clearly demonstrate the superiority our proposed methods over RS, reducing both construction time and saving orders of magnitude in terms of both query time and size. We cannot achieve higher precision for RS when the observed error is smaller than 10^{-5}, since in those cases, ε is small enough that the random sample size is as big as the



(a) Construction time. (b) Communication. (c) Size. (d) Query time.

Figure 9: Effect of varying measured *err* on RS, SS on one-dimensional data.

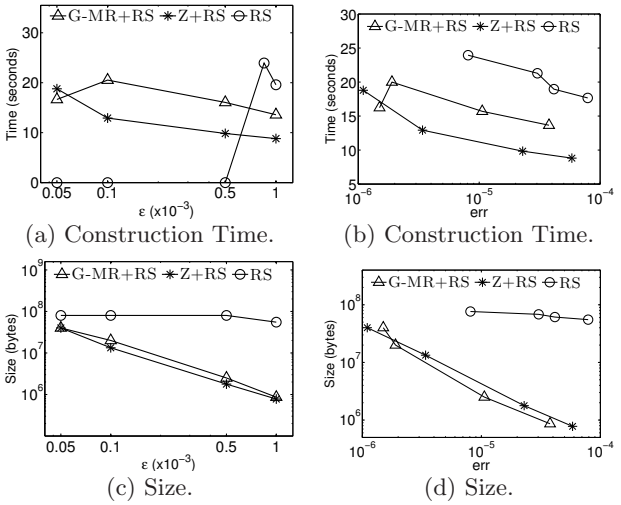
size of the original dataset (i.e., KDE from a random sample consists of the entire original dataset, leading to no error).

6.4 One Dimension: Parallel and Distributed

Default setup. We now shift our attention in 1d. The default data set is *Meme* with 50 million records, $\delta = 0.001$, and σ at 1 day, over 273 days of data. Since GS (grouping selection) is a complementary technique that works with any other method, we focus on RS and SS (sort selection). We only report the results from the parallel and distributed setting. The trends in the centralized setting are similar.

SS vs. RS. By varying the observed error *err*, Figure 9 compares the two methods across different metrics. To achieve smaller observed errors in constructing KDE_Q, SS is more efficient as shown in Figure 9(a), but RS becomes faster for larger observed errors. A similar trend is observed for the communication cost in Figure 9(b). In terms of reducing the size of Q, SS does a much better job than RS as shown in Figure 9(c), $|Q|$ in SS is 1-4 orders of magnitude smaller than $|Q|$ in RS, the gap is particularly large for smaller observed errors. This translates to the query time in Figure 9(d), where evaluating $\text{KDE}_Q(x)$ using Q produced by SS is much faster than doing so over Q from RS for most observed errors. When *err* becomes large, around 10^{-2}, the RS method catches up, corresponding to the sizes of Q becoming closer.

In conclusion, SS in general performs better than or comparable to RS on observed error. But it has much better theoretical guarantees in size and query time as shown in Section 4 ($\frac{1}{\varepsilon}$ vs. $O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta})$), which is critical in practice since users generally use a method by setting an ε value.



(a) Construction Time. (b) Construction Time.

(c) Size. (d) Size.

Figure 8: Effect of ε and ℓ_∞ error *err* on G-MR+RS, Z+RS, RS for high precision case.

7. CONCLUSION

This paper presents the new state-of-the-art for computing kernel density estimates over enormous real world data sets (up to 100 million points). These are essential statistical tools for large-scale data analysis and physical simulations. Our methods produce a coreset representation of the data which can be used as proxy for the true data while guaranteeing approximation error on size and runtime. Moreover, an extensive experimental study demonstrates that our methods provide clear improvements over existing techniques in terms of size, accuracy, and runtime. Interesting open problems include extending to kernel approximation on complex data types, such as graph and string kernels.

8. REFERENCES

[1] A. W. Appel. An efficient program for many-body simulation. *SIAM J.Scientific and Statistical Computing*, 6:85–103, 1985.

[2] N. Aronszajn. Theory of reproducing kernels. *Transactions AMS*, 68:337–404, 1950.

[3] A. Bagchi, A. Chaudhary, D. Eppstein, and M. T. Goodrich. Deterministic sampling and range counting in geometric data streams. *ACM Transactions on Algorithms*, 3:16, 2007.

[4] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest-pair and n-body potential fields. In *SODA*, 1995.

[5] B. Chazelle. *The Discrepancy Method*. Cambridge, 2000.

[6] B. Chazelle and J. Matousek. On linear-time deterministic algorithms for optimization problems in fixed dimensions. *Journal of Algorithms*, 21:579–597, 1996.

[7] Y. Chen, M. Welling, and A. Smola. Super-samples from kernel hearding. In *Conference on Uncertainty in Artificial Intellegence*, 2010.

[8] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *PODS*, 2006.

[9] G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang. Optimal sampling from distributed streams. In *PODS*, 2010.

[10] R. Duan, S. Pettie, and H.-H. Su. Scaling algorithms for approximate and exact maximum weight matching. Technical report, arXiv:1112.0790, 2011.

[11] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[12] S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *ICDT*, pages 143–157, 2003.

[13] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, 1996.

[14] L. Greengard and J. Strain. The fast Gauss transform. *Journal Scientific and Statistical Computing*, 12, 1991.

[15] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, 2001.

[16] S. Guha, N. Koudas, and K. Shim. Approximation and streaming algorithms for histogram construction problems. *ACM TODS*, 31:396–438, 2006.

[17] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *SIGMOD*, pages 463–474, 2000.

[18] S. Har-Peled. *Geometric Approximation Algorithms*. Chapter 2. American Mathematical Society, 2011.

[19] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ICDE*, 1997.

[20] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *SIGMOD*, 1997.

[21] Z. Huang, L. Wang, K. Yi, and Y. Liu. Sampling based algorithms for quantile computation in sensor networks. In *SIGMOD*, pages 745–756, 2011.
[22] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, 1998.
[23] M. C. Jones, J. S. Marron, and S. J. Sheather. A brief survey of bandwidth selection for density esimation. *American Statistical Association*, 91:401–407, 1996.
[24] S. Joshi, R. V. Kommaraju, J. M. Phillips, and S. Venkatasubramanian. Comparing distributions and shapes using the kernel distance. In *ACM SoCG*, 2011.
[25] V. Kolmogorov. BLOSSOM V: A new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming*, 1(1), 2009.
[26] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal histograms for hierarchical range queries. In *PODS*, 2000.
[27] X. Lin, J. Xu, Q. Zhang, H. Lu, J. X. Yu, X. Zhou, and Y. Yuan. Approximate processing of massive continuous quantile queries over high-speed data streams. *TKDE*, 18(5):683–698, 2006.
[28] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD*, pages 426–435, 1998.
[29] J. Matoušek. Approximations and optimal geometric divide-and-conquer. In *STOC*, 1991.
[30] J. Matoušek. *Geometric Discrepancy*. Springer, 1999.
[31] E. Parzen. On estimation of a probability density function and mode. *Annals of Mathematical Statistics*, 33:1065–1076, 1962.
[32] E. Parzen. Probability density functionals and reproducing kernel Hilbert spaces. In *Proceedings of the Symposium on Time Series Analysis*, pages 155–169, 1963.
[33] J. M. Phillips. ε-samples for kernels. In *SODA*, 2013.
[34] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, pages 294–305, 1996.
[35] V. C. Raykar and R. Duraiswami. Fast optimal bandwidth selection for kernel density estimation. In *SDM*, 2006.
[36] V. C. Raykar, R. Duraiswami, and L. H. Zhao. Fast computation of kernel estimators. *J. of Computational and Graphical Statistics*, 19(1):205–220, 2010.
[37] M. Rosenblatt. Remarks on some nonparametric estimates of a density function. *A. Math. Stat.*, 27:832–837, 1956.
[38] B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2002.
[39] D. W. Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley, 1992.
[40] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.
[41] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10:2615–2637, 2009.
[42] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, 1986.
[43] S. Suri, C. D. Tóth, and Y. Zhou. Range counting over multidimensional data streams. *Discrete and Computational Geometry*, 36:633–655, 2006.
[44] B. A. Turlach. Bandwidth selection in kernel density estimation: A review. Discussion paper 9317, Istitut de Statistique, UCL, Louvain-la-Neuve, Belgium.
[45] K. R. Varadarajan. A divide-and-conquer algorithm for min-cost perfect matching in the plane. In *FOCS*, 1998.
[46] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.
[47] J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *CIKM*, 1998.
[48] K. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. Smola. Feature hashing for large scale multitask learning. In *ICML*, 2009.
[49] C. Yang, R. Duraiswami, and L. S. Davis. Efficient kernel machines using the improved fast gauss transform. In *NIPS*, 2004.
[50] C. Yang, R. Duraiswami, N. A. Gumerov, and L. Davis. Improved fast gauss transform and efficient kernel density estimation. In *ICCV*, 2003.
[51] T. Zhang, R. Ramakrishnan, and M. Livny. Fast density estimation using cf-kernel for very large databases. In *KDD*, pages 312–316, 1999.

9. APPENDIX

PROOF. OF LEMMA 2: We use a result from Joshi *et.al.* [24] (Theorem 5.1) that states that if $Q \subset P$ satisfies the following property (actually something more general), then $\|\text{KDE}_P - \text{KDE}_Q\|_\infty \leq \alpha$. For *any* interval $I = [a, b]$ we must have $||Q \cap I|/|Q| - |P \cap I|/|P|| \leq \alpha$. So we just need to show that the interval property is satisfied with $\alpha = 2\varepsilon$.

First note that any set P_j that is either entirely not in I (that is $|I \cap P_j| = 0$) or in I (that is $|P_j \cap I| = |P_j|$) contributes to $|Q \cap I|/|Q|$ the same as $|P \cap I|/|P|$ and this has no effect on the difference between the two. Since the sets are sorted, there are at most two sets $P_{j'}$ and $P_{j''}$ which contribute to the interval query error, depending on if $q_{j'}$ and $q_{j''}$ are in I or not. Since $|P_j|/|P|$ and $|\{q_j\}|/|Q|$ are at most ε, then the total error of leaving $q_{j'}$ and $q_{j''}$ in or out is at most $\alpha = \varepsilon + \varepsilon = 2\varepsilon$.

To see the tighter bound, note that if $q_{j'}$ (resp. $q_{j''}$) is in I, then at least half of $P_{j'}$ (resp. $P_{j''}$) is also in I. Thus each can contribute at most $\varepsilon/2$ to the error, leading to total error $\alpha = \varepsilon/2 + \varepsilon/2 = \varepsilon$. □

PROOF. OF THEOREM 2: MergeReduce takes $O(\log \frac{n}{k})$ rounds. In the ith round, we have $\frac{n}{2^{i-1}k}$ sets of points and each set has k points. We arbitrarily pair two sets, and run the Blossom matching algorithm over the union of $2k$ points. This step takes $O(\frac{n}{2^i k}(2k)^3)$ time. Note that the second part in a reduce step takes only linear time. Hence, the total running time is $\sum_{i=1}^{\log \frac{n}{k}} \frac{n}{2^i k}(2k)^3$ which is $O(nk^2)$.

The main result of [33] shows that each round on $n_i = n/2^{i-1}$ points produces a $O(\sqrt{\log n_i}/n_i)$-approximate KDE of the previous round, so after $\log \frac{n}{k}$ rounds the error is

$$\sum_{i=1}^{\log(n/k)} \frac{\sqrt{\log(n/2^i)}}{n/2^i} = O\left(\frac{1}{k} \log \frac{n}{k} \sqrt{\log k}\right)$$

with constant probability. Setting $k = O(\frac{1}{\varepsilon} \log n \sqrt{\log \frac{1}{\varepsilon}})$ achieves ε error total and a runtime of $O(\frac{n}{\varepsilon^2} \log^2 n \log \frac{1}{\varepsilon})$.

Sampling $O(\frac{1}{\varepsilon^2})$ points first takes $O(n)$ time and effectively sets $n = \frac{1}{\varepsilon^2}$ in the bound above. The with-sampling result follows. □

PROOF. OF THEOREM 3: MergeReduce has $O(\log \frac{n}{k})$ rounds. In the ith round, we have $\frac{n}{2^{i-1}k}$ sets of points and each set has k points. We arbitrarily pair all sets and for each pair run Grid over the union of $2k$ points, in total $O(\frac{n}{2^i k}(2k) \log \frac{1}{\varepsilon})$ time. Note that the second part in a reduce step takes only linear time. Hence, the total running time is $\sum_{i=1}^{\log \frac{n}{k}} \frac{n}{2^i k}(2k) \cdot \log(1/\varepsilon)$ which is $O(n \log(1/\varepsilon))$.

The main result of [33] (now adjusting for C_M) shows that each round on $n_i = n/2^{i-1}$ points produces a $O(C_M \sqrt{\log n_i}/n_i)$-approximate KDE of the previous round, so after $\log \frac{n}{k}$ rounds the approximation error is

$$\sum_{i=1}^{\log(n/k)} \frac{C_M \sqrt{\log(n/2^i)}}{n/2^i} = O\left(\frac{C_M}{k} \log \frac{n}{k} \sqrt{\log k}\right)$$

with constant probability. Since by Lemma 4 we have $C_M = O(\log(1/\varepsilon))$, setting $k = O(\frac{1}{\varepsilon} \log n \log^{1.5} \frac{1}{\varepsilon})$ achieves ε error total and a runtime of $O(n \log \frac{1}{\varepsilon})$ using Grid-MR.

Sampling $O(\frac{1}{\varepsilon^2})$ points first takes $O(n)$ time and effectively sets $n = \frac{1}{\varepsilon^2}$ in the bound above. The Grid-MR +RS result follows. □