Bandlimited OLAP Cubes for Interactive Big Data Visualization

Caleb Reach[*] Chris North[†]

Virginia Tech

ABSTRACT

Visualizations backed by data cubes can scale to massive datasets while remaining interactive. However, the use of data cubes introduces artifacts, causing these visualizations to appear noisy at best and deceptive at worst. Moreover, data cubes highly constrain the space of possible visualizations. For example, a histogram backed by a data cube is constrained to have a bin width that is a multiple of the data cube bin size. Similarly, for dynamic queries backed by data cubes, query extents must be aligned with bin boundaries. We present bandlimited OLAP (online analytical processing) cubes (BLOCs), a technique that uses established tools from digital signal processing to generate interactive visualizations of very large datasets. Based on kernel density plots and Gaussian filtering, BLOCs suppress the artifacts that occur in data cubes and allow for a continuous range of zoom/pan positions and continuous dynamic queries.

1 INTRODUCTION

Interactive visualization tools that render visualizations by iterating over each point in the dataset cannot handle large datasets, as this iteration becomes prohibitively expensive. This problem may be addressed by choosing a small random sample of the dataset for visualization purposes; however, such a sample may omit important outliers. For example, an event that occurs once every 10,000 data points will likely be missed in a 1000 point random sample.

Data cubes address the interactive big data visualization problem by binning all data points in a preprocessing step in which each point in the dataset is assigned to a bin based on the values of one or more attributes. This mapping between attribute values and bins depends on the the resolution of the data cube, which is inversely proportional to the size of the bins. A high resolution data cube with small bin sizes can answer more precise queries and produce more precise visualizations than a low resolution data cube with large bin sizes. Resolution is task-dependent and is often chosen by hand.

A key advantage of data cubes is that their storage requirements depend on their resolution rather than on the size of the source dataset, and once constructed, a data cube can be used to quickly answer visualization queries. For example, ImMens [16] demonstrates that data cubes can allow a modern laptop to interactively visualize one billion points at interactive rates.

However, data cubes are a fundamentally discrete representation of a dataset, and as such, they constrain the space of possible visualizations to discrete forms such as histograms and binned heatmaps. Such discretization introduces distracting, and potentially deceptive, aliasing artifacts [18]. Aliasing in visualization may be classified into two categories: spatial aliasing, which occurs when histogram or heatmap bins appear to indicate patterns that do not exist, and temporal aliasing, which occurs when an interactive or animation visualization misrepresents patterns in the source dataset.

[*]e-mail: caleb.reach@vt.edu
[†]e-mail: north@cs.vt.edu

IEEE Symposium on Large Data Analysis and Visualization 2015
October 25–30, Chicago, Il, USA
978-1-4673-8517-6/15/$31.00 ©2015 IEEE

For example, Figure 1 shows a simple sales dataset that exhibits aliasing artifacts when visualized with a monthly histogram. When the frequency of data cube binning does not appropriately match the frequency of patterns in the dataset, spurious aliased frequency patterns emerge. Since most datasets contain complex patterns, they will exhibit aliasing artifacts when visualized with a histogram regardless of bin width. Data cubes cannot be used to generate more natural continuous representations, such as kernel density estimation plots, that would more faithfully represent the distribution.

Further, interactive navigation and dynamic querying of the data is critical to enabling exploration of big data, but data cubes limit interaction to the pre-computed discrete boundaries. For example, the bin width and bin positions of the data cube constrain the bin width of positions of the histogram. While multiple data cubes at different resolutions can enable simple discrete zooming, data cubes cannot provide smooth continuous zooming and navigation without further amplified aliasing. Likewise, when a data cube is used to implement dynamic queries or brushing, the query or brush must be box-shaped and positioned only on bin boundaries. This results in significant temporal aliasing when dragging a brush or query widget.

Although Lagrange interpolation can be used to draw a smooth curve passing through histogram bins or to provide values for in-between brush positions, these in-between values don't have a simple, intuitive meaning in terms of the source dataset, as aliasing has already been baked into the binning. Lagrange interpolation acts as a anti-imaging lowpass filter [20], but it does not remove the aliasing introduced by binning.

To address these problems, we introduce a new approach that applies well-established signal processing techniques to information visualization. We avoid aliasing by filtering before binning the dataset, and we allow continuous visualizations using the concept of bandlimited interpolation. Our contributions are as follows:

1. We introduce Bandlimited OLAP Cubes (BLOCs), a technique for generating artifact-free interactive visualizations of large datasets. Visualizations backed by BLOCs can support brushing, dynamic querying, and zooming in realtime (60 FPS). Our technique allows continuous brush positions and continous zooming.

2. We show that the assymetry between brush resolution and graphic resolution can be exploited to reduce data cube storage requirements.

BLOCs have two primary advantages over data cubes due to their use of more advanced signal processing theory. First, BLOCs suppress aliasing, providing more accurate visualizations. Secondly, BLOCs support continuous interaction. While the storage requirements for BLOCs are similar to the storage requirements for data cubes, BLOCs support an infinite number of in-between zoom levels and in-between brush positions.

2 BANDLIMITED DATA CUBES

BLOCs are based heavily on digital signal processing techniques. We first review some of these techniques.

2.1 Background

The Fourier transform decomposes a signal into complex sinusoids. More precisely, the Fourier transform maps a signal $x(t)$ from the time domain to a complex-valued signal $X(\omega)$ in the frequency domain. For real-valued $x(t)$, the corresponding frequency-domain signal has the property that each negative frequency is the complex conjugate of the corresponding positive frequency. Because the negative frequencies add no new information, $X(\omega)$ is often treated as containing positive frequencies only.

For a periodic signal, the sinusoids into which the time-domain signal is decomposed are harmonically related such that the frequency of each sinusoid is a multiple of the fundamental frequency (the frequency corresponding to the period of the signal). Since a finite-length signal can be encoded as a periodic signal, a finite-length signal can be decomposed into a sum of harmonically-related sinusoids.

The Nyquist–Shannon sampling theorem states that a signal containing no frequencies higher than f_n is uniquely determined by a sampling at sampling rate $f_s = 2f_n$. The frequency $f_n = f_s/2$ is known as the Nyquist frequency. If $X(\omega)$ is compactly supported (i.e. is zero outside of some bounded interval) then the signal is said to be bandlimited. A finite-length discrete (sampled) bandlimited time-domain signal can thus be represented as a finite-length discrete complex frequency-domain signal.

The Dirac delta function[1] $\delta(t)$ can be seen as the derivative of the unit step function or as the Gaussian probability density function in the limit as the standard deviation goes to zero. In the frequency domain, the Dirac delta contains all frequencies with equal magnitude.

An impulse train (also known as a Dirac comb) is a signal that contains equally spaced Dirac deltas. The spacing of the Dirac deltas is the period T of the impulse train, and the train is infinitely long in both directions. The Fourier transform of an impulse train is an impulse train with the reciprocal period $1/T$. The sampling of a continuous signal is often modeled as time-domain multiplication with an impulse train. This model shows why aliasing occurs: multiplication in the time domain corresponds to convolution in the frequency domain, and convolving a signal with an impulse train results in equally spaced copies of the signal. When the signal has not been appropriately bandlimited, the copies overlap, causing aliasing.

Guassian filters (which are used for blurring in image processing) are filters with a Gaussian impulse response (called a kernel in image processing). Interestingly, the frequency response of a Gaussian filter is also Gaussian. Since the Gaussian function approaches zero rapidly outside its central lobe, it can be regarded as approximately finite length and approximately bandlimited and can thus be very closely approximated by a discrete finite impulse response (FIR) filter. Because a Gaussian kernel is symmetric, multiplications can be reduced by half in a direct implementation of the filter. For example, a zero-phase Gaussian filter can be implemented as

$$y[n] = a_0 x[n] + \sum_{i=1}^{k} a_i (x[n+i] + x[n-i]),$$

where $x[n]$ is the input signal, a_0 is the center coefficient of the filter, and $a_1, a_2, \ldots, a_k$ are the filter coefficients to the right of the center coefficient.

Since the convolution of two Gaussian functions is a Gaussian function [3], the effect of two Gaussian filters in serial can be produced by a single Gaussian filter. The variance of the resultant Gaussian is the sum of the variances of the source Gaussians. Thus the standard deviation σ_c of the resultant Gaussian is given by

$$\sigma_c = \sqrt{\sigma_a^2 + \sigma_b^2}$$

where σ_a and σ_b are the standard deviations of the serial Gaussian filters.

The B-Spline kernels are given by successive convolutions of the boxcar function. The polynomial bandlimited impulse train algorithm (polyBLIT) [17] uses B-Spline interpolators to generate bandlimited impulse trains, and we use the same technique to perform bandlimited binning. The first several kernels are repeated here. The zeroth-order kernel b_0 is simply the box kernel. The next kernel, b_1, is the triangle kernel. The next kernel is

$$b_2(t) = \begin{cases} \frac{1}{2}\left(\frac{t}{T_s} + \frac{3}{2}\right)^2 & -\frac{3}{2} \le \frac{t}{T_s} < -\frac{1}{2} \\ \frac{3}{4} - \left(\frac{t}{T_s}\right)^2 & -\frac{1}{2} \le \frac{t}{T_s} < \frac{1}{2} \\ \frac{1}{2}\left(\frac{t}{T_s} - \frac{3}{2}\right)^2 & \frac{1}{2} \le \frac{t}{T_s} < \frac{3}{2} \\ 0 & \text{otherwise} \end{cases}$$

where T_s is the sampling period. We use the third-order B-Spline kernel for binning and interpolation, which is given as

$$b_3(t) = \frac{1}{6} \begin{cases} \left(2 + \frac{t}{T_s}\right)^3 & -2 \le \frac{t}{T_s} < -1 \\ 4 - 3\left(\frac{t}{T_s}\right)^2 \left(2 + \frac{t}{T_s}\right) & -1 \le \frac{t}{T_s} < 0 \\ 4 + 3\left(\frac{t}{T_s} - 2\right)\left(\frac{t}{T_s}\right)^2 & 0 \le \frac{t}{T_s} < 1 \\ -\left(\frac{t}{T_s}\right)^3 & 1 \le \frac{t}{T_s} < 2 \\ 0 & \text{otherwise} \end{cases}$$

The B-Spline kernels approach the Gaussian in the limit as the order goes to infinity. Convolving a low-order B-Spline kernel with a Gaussian gives a very accurate approximation of the Gaussian.

2.2 Method

Our goal is to visualize and interact with a dataset in a flexible manner that does not introduce aliasing artifacts. BLOCs are a very general technique in support of this goal. A BLOC is the data structure constructed by any technique that satisfies the following:

1. The technique begins with a dataset signal, which is an infinitely precise and perfectly overfitted density function that will be formally defined later in this section.
2. The technique filters this dataset signal to bandlimit it and prevent aliasing.
3. The technique samples and stores this bandlimited signal.
4. The technique responds to queries using this sampled signal, filtering it to avoid imaging.

BLOCs can be used to implement kernel density estimation, kernel smoothing, brushing, and dynamic queries, provided that the kernel used in all cases is bandlimited. Although BLOCs as thus defined are very general, we also introduce a specific BLOC-based technique in this paper to motivate the use of BLOCs and to begin to explore the BLOC design space.

Our goal for this specific technique is to produce a visualization of a dataset that suppresses spatial aliasing and allows dynamic querying in a way that suppresses temporal aliasing. Histograms exhibit severe aliasing artifacts and therefore cannot be used to satisfy this goal. However, kernel density plots do suppress spatial

[1] Although the Dirac delta is technically not a function, it is often treated as one in engineering contexts.

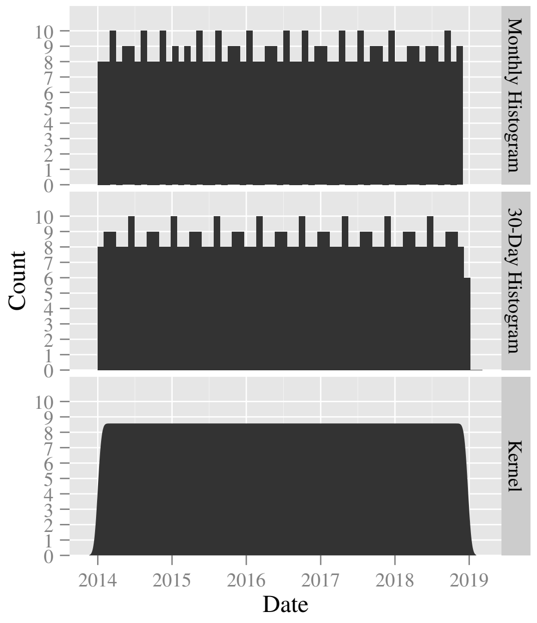

Figure 1: Histograms compared to kernel plots. The input is a very simple synthetic sales dataset that contains one sale per weekday and no sales on the weekends. The histograms show a deceptive 20% variation in monthly sales, even though the sales pattern for each week is identical. This variation exhists regardless of whether bins are month-aligned or 30 days wide. The kernel plot accurately shows that the sales pattern is constant at the monthly level.

aliasing when an appropriate kernel is used. Our technique uses the Gaussian kernel, which offers excellent alias suppression.

For dynamic queries, a commonly used scheme is to have a range selector for one or more dimensions. The start and end points of this range selector can be dragged individually, and dragging the center of the range moves the start and end points together, allowing the range to be swept across the extents of the dimension. However, such a scheme exhibits temporal aliasing artifacts when the range is swept quickly and when the start and end points of the range are constrained to pixel positions.

Our scheme instead uses Gaussian selectors along one or more dimensions. For each dimension, the user specifies a center and standard deviation. Instead of a strict subset, data points are weighted based on their distance from the selector center using the Gaussian function given by the chosen standard deviation. Given that Gaussian functions exhibit low-pass characteristics, this naturally suppresses temporal aliasing as the center is dragged.

Thus, our technique can be seen conceptually as follows:

1. Weight each point based on the given Gaussian selectors.

2. Generate a kernel density plot where each point contributes to the plot proportionally to its weight.

A naive implementation would iterate over each point in the dataset, assigning a weight to each point, and then generate a kernel density plot. However, such an approach would not scale well to large datasets.

Our approach has two phases: preprocessing and interactive rendering. The preprocessing steps are conceptually as follows:

1. Construct the dataset signal $S(\mathbf{x})$ from the source dataset.

2. Filter this dataset signal using B-Spline anti-aliasing filters.

3. Sample the filtered signal.

4. Construct a mipmap from this filtered signal.

In practice, the first three steps are combined into a single anti-aliased binning step. The steps for interactive rendering occur each frame and are as follows:

1. Select an appropriate mipmap level.

2. Collapse this mipmap level's discrete signal based on the Gaussian selectors, reducing the dimensionality of the signal.

3. Run a matching filter to ensure that the final signal will have the desired target standard deviation. This matching filter is a Gaussian discrete FIR filter.

4. Filter this discrete signal using B-Spline anti-imaging filters.

5. Sample this signal at the desired target sampling rate.

In practice, the last two steps are combined into a single anti-aliased interpolation step. The whole procedure is summarized in Figure 2.

The Dataset Signal

The first step is to place all dataset points into a dataset signal, which maps from k-dimensional attribute space to density. This dataset signal is zero at all locations where there are no data points and infinity at the locations of data points. For an input dataset $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n$, where each data point is represented by a k-dimensional vector, the dataset signal $S(\mathbf{x})$ is given as

$$S(\mathbf{x}) = \sum_{i=1}^{n} \delta(\mathbf{x} - \mathbf{p}_i).$$

The dataset signal can be seen as the empirical density function of the dataset multiplied by the number of points in the dataset. To see what this signal represents, suppose we have a simple access log dataset where each log entry is a timestamp. A 1D data cube can be used to quickly visualize this dataset, where each bin in the data cube stores the number of accesses that occurred within its time range. This data cube can be represented as a function $f(t)$ mapping from time t to access log rates, which are given by the number of points in the bin containing the given time divided by the bin's duration.

Using this function, the number of accesses occurring within some time range is given by integrating $f(t)$ over the time range. This gives the correct result in all cases where the query time range is aligned with bin boundaries, i.e. when neither the start time nor end time cut through the middle of a bin. In order to admit more precise query regions, the data cube must be constructed with a higher resolution, resulting in bins with shorter durations. In the limit as the data cube resolution goes to infinity and bin durations go to zero, the function $f(t)$ approaches the dataset signal $S(t)$, as illustrated in Figure 3. Whereas there are many datasets that can produce a given data cube, there is a one-to-one mapping between datasets and dataset signals. Furthermore, integrating the dataset signal over any region correctly gives the number of points contained within that region. The dataset signal described in the preceding paragraph $S(\mathbf{x})$ is a simply a multidimensional generalization of this idea.

B-Spline Binning

This dataset signal is then discretized using B-Spline anti-aliasing filters to bin the data points, as illustrated in Figure 4. This anti-aliasing filter distinguishes BLOCs from traditional data cubes: in traditional data cubes, each point is assigned to a single bin,

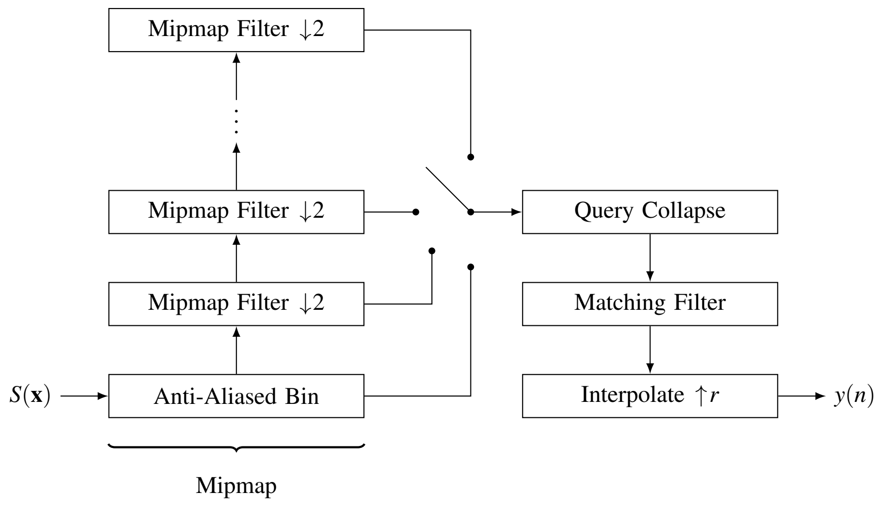

Figure 2: Overview of mipmap creation. Here, r is f_t/f_s, where f_s is the sampling rate of the mipmap level and f_t is the target sampling rate. Both the mipmap filters and the matching filter are Gaussian FIR filters. Both the anti-aliased binning and the interpolation steps use third order B-Spline filters.

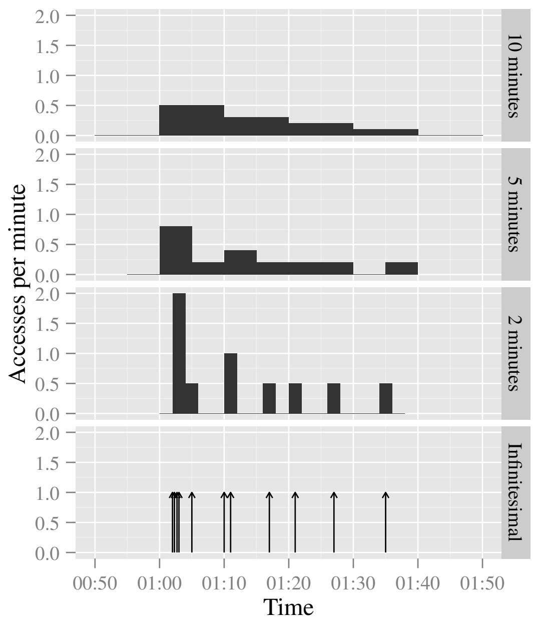

Figure 3: The data signal can be seen as an infinitely precise data cube. Here, the histograms show the density of points for each bin. As the bin size decreases, the densities more accurately represent the source dataset. In the limit as the bin size approaches zero, the density function perfectly represents the source dataset and is equivalent to the data signal. The arrows in this final plot represent the Dirac delta impulses.

whereas in a BLOC, a single point is distributed across a neighborhood of bins. In a traditional data cube, a point at the left end of a bin is indistinguishable from a point on the right end of a bin once the data cube has been constructed. In a BLOC, however, a point on the left end of a bin affects bins to the left more than a point on the right end of a bin.

Conceptually, discretizing using B-Spline anti-aliasing filters involves convolving the dataset with the B-Spline kernel and then sampling by multiplying with an impulse train. The period of this train is given by the resolution of the BLOC. In practice, however, B-Spline binning involves iterating over each point and increasing the values in a small neighborhood of bins near the point. The amount that each value is increased is given by the B-Spline kernel. Once all points have been binned, this binning forms the lowest (finest) level of a mipmap [1].

Each level of the mipmap has an associated sampling rate f_s and standard deviation σ. The sampling rate for this lowest level is given by the BLOC resolution. To determine the standard deviation associated with a B-Spline filter, we used a computer algebra system to perform a least-squares fit between the B-Spline kernel and the Gaussian probability density function. We found that the minimizing standard deviation for the third order B-Spline function was 0.596553. Given that the B-Spline kernel is scaled based on the standard deviation, this gives a standard deviation of $\sigma = 0.596553/f_s$.

Mipmap construction

To generate higher (courser) levels, the base level is successively filtered using a Gaussian filter and downsampled by a factor of two. The Gaussian filter is designed to have a standard deviation of two samples at the downsampled rate, as this effectively bandlimits the signal [21]. Although any downsampling factor could be used in theory, a factor of two represents a good trade-off between space requirements, which are higher for lower downsampling factors, and rendering time requirements, which are higher for higher downsampling factors because the matching filter must be more powerful.

The sampling rate for all mipmap levels except for the lowest is half of the sampling rate of the level beneath it. The standard deviation for each mipmap level except for the lowest is given by the cumulative effect of the B-Spline binning filter, the mipmap filters for all lower layers, and the B-Spline interpolation filter, which has a standard deviation of $0.596553/f_s$, where f_s is the sampling rate

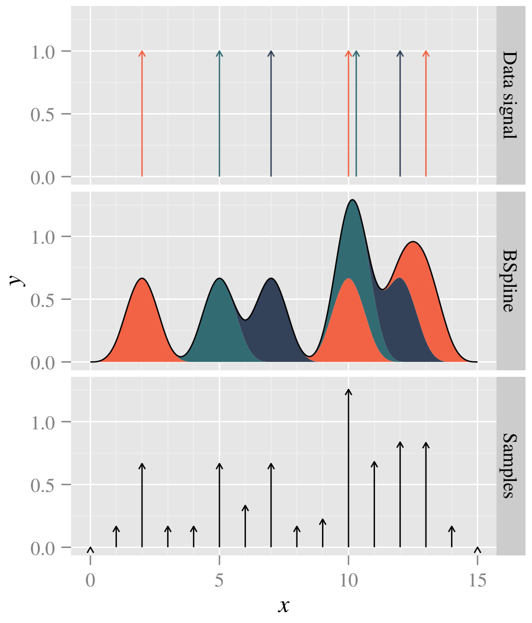

Figure 4: Discretizing the data signal using B-Spline binning, which is based on the PolyBLIT algorithm [17]. Conceptually, the data signal is convolved with a B-Spline filter and then sampled. In practice, each point in the dataset is directly binned into a neighborhood of bins. The top subplot shows the data signal comprising Dirac delta impulses. The middle subplot shows each impulse replaced with a B-Spline kernel and arranged in a stacked graph formation. The black outline of the middle subplot is the result of the convolution. The signal in the bottom subplot is given by convolving this result with an impulse train.

of the mipmap level. The variance σ_i^2 for the current level is found be adding the variance σ_{i-1}^2 of the level below the current level and the B-Spline filter standard deviation $0.596553/f_s$. The standard deviation is then simply the square root of this variance.

Mipmap level selection

To render at a desired sample rate f_t and a target standard deviation σ_T, a mipmap level is selected that can satisfy σ_T. A mipmap level satisfies a given standard deviation if its associated standard deviation is small enough that a matching filter can be designed to accomplish this goal.

Matching filter

The selected mipmap level is run through a matching filter and an interpolation filter. The matching filter is a Gaussian FIR filter that is designed to bring the final cumulative standard deviation to the given target standard deviation. For a mipmap level with standard deviation σ_i and a target standard deviation σ_T, the matching filter's standard deviation σ_M is given by

$$\sigma_M = \sqrt{\sigma_T^2 - \sigma_i^2}.$$

In our tests, the result was acceptable as long as the matching filter has a standard deviation greater than one. A more conservative choice would be to require the matching filter to have a standard deviation of at least two to ensure that the coefficients are bandlimited. To be even more conservative, the matching filter could have a polyphase implementation that upsamples by a factor of two to ease the burden of the B-Spline interpolation filter to remove imaging.

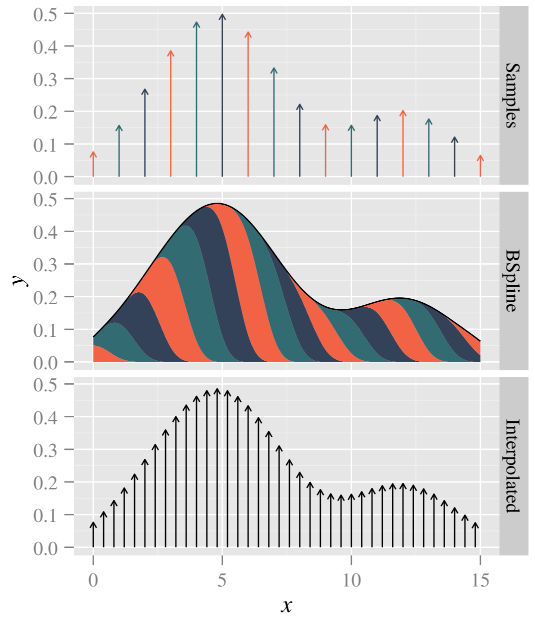

Figure 5: After the matching filter, the data must be interpolated to convert to the target sampling rate. Although the use of B-Spline filters for interpolation causes the interpolated curve to not pass through the source points, this effect is anticipated and accounted for in the calculation of the matching filter's standard deviation.

B-Spline Interpolation

The interpolation filter is a simple B-Spline filter, and the interpolation process is shown in Figure 5.

2.3 Analysis

Space complexity is determined by the resolution of the BLOC. The space complexity of the initial binning is given by the product of the number of samples in each dimension. For an application where we wish to show 1D kernel density plots for each of k dimensions and support dynamic filtering along these dimensions, we can generate k BLOCs, each of which is high-resolution in the dimension of the kernel density plot and low-resolution in all other dimensions. The space complexity is then given by the kernel density plot resolution p and the brushing resolution b as $O(kpb^{k-1})$. Because the brushing resolution is generally much lower than the kernel density plot resolution, this asymmetric approach represents a significant savings over the alternative approach of storing a single high-resolution data cube, which has a space complexity of $O(p^k)$. The time complexity for binning a set of such asymmetric BLOCs is $O(kpb^{k-1} + 4^k n)$, where n is the number of data points in the dataset. The base of the 4^k component can be reduced by using lower-order B-Spline kernels at the expense of increased aliasing, but given that data cubes only practical when low-dimensional, this should not be necessary. Binning is readily parallelized and can be implemented using Map Reduce [8].

The number of samples required for a mipmap level l_n is based on the number of samples of the mipmap level below l_{n-1}. For a mipmap filter with s coefficients, we have

$$l_n = (l_{n-1} + s)/2.$$

The storage requirements L for the whole mipmap is given by the sum of the lengths of the mipmap levels:

$$L = \sum_{i=0}^{n} l_i$$

Solving the recurrence relation and the sum gives

$$L = (2 - 2^{-i})b + (2^{-i} + i - 1)s.$$

The first component, $(2 - 2^{-i})b$, approaches $2b$ as i goes to infinity. The second component approaches is as i goes to infinity. However, in practice, we stop producing mipmap levels before the second term becomes problematic. With this stipulation, adding a mipmap for k dimensions requires roughly 2^k times the storage of the base binning.

3 Implementation

We built two implementations: one that generates a two-dimensional mipmapped bandlimited cube using a B-Spline 3 kernel and one that generates a non-mipmapped three-dimensional bandlimited cube using a B-Spline 2 kernel. Both were tested with the OpenStreetMap dataset [12], and the two-dimensional cube was additionally tested using a synthetic dataset.

The 2D cube uses a C++ back-end and Javascript front-end. For the OpenStreetMap dataset, the C++ back-end use the Osmium library to process a dump of OpenStreetMap data in Protocol Buffer format and generate the cube. The attributes used to generate the cube are longitude and time, and the cube has a size of 250,000 bins in the time dimension and 38 bins in the longitude dimension. The back-end processes just over three billion points in just over eight minutes on a late 2013 MacBook Pro. The front-end performs the mipmap generation and rendering and supports zooming and dynamic querying using Gaussian queries. Rendering is performed using the canvas element, and each frame is rendered within 1-3 milliseconds except for the first, which takes between 8 and 40 milliseconds. The initial page load takes seven seconds.

The synthetic dataset uses a data cube with a significantly lower resolution in the time axis (12500), but the render-time performance is similar to the OpenStreetMap cube. This is unsurprising, since mipmapping allows an appropriately-sized cube to be selected for each frame. The back-end for the synthetic dataset can bin roughly 50 million points per second using a single core. The initial page load completes in just under three seconds, which is also faster than the larger cube.

Because the initial 2D binning for our example is high-resolution along the mipmapped dimension and low-resolution along the brushed dimension, we use a tall-and-skinny memory layout for the initial binning to improve cache performance. In this layout, adjacent samples along the brushed dimension are stored in adjacent locations in memory. Once binning is complete, we transpose this representation, giving a short-and-fat layout such that adjacent samples along the mipmapped dimension are stored in adjacent locations in memory. This should improve caching performance for mipmap computation, where filtering processes a sliding window of samples along the brushed dimension. We should note that we did not measure the performance of an alternative implementation that does not take locality into account, and additionally, a GPU implementation would likely be less sensitive to locality issues.

In our current implementation, the binned BLOC is transferred in its entirety to the client. The client then computes all higher mipmap layers from this base layer. In production, the server would instead generate the full mipmap and then send tiles to the client on-demand in the manner of mapping applications. In this scheme, each mipmap level would be broken up into tiles such that each level of zoom can be rendered using a small number of tiles.

For the 3D cube, the binning code uses C++ and processes all OpenStreetMap nodes in eleven minutes and constructs a 3D data

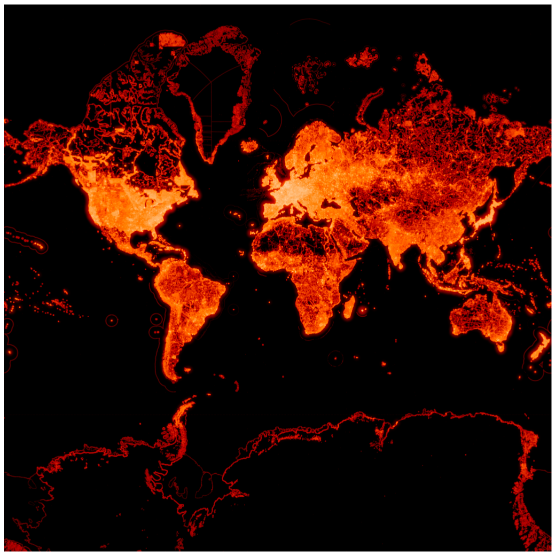

Figure 6: An example of a visualization generated using BLOCs. This visualization adds a small amount of a Gaussian blurred version of the image to the original density function, and then uses log scaling of density followed by tanh saturation and color mapping to produce the final image. The addition of Gaussian blurred version produces a glare effect that emphasizes regions of high density.

cube with a spatial resolution of 1024 by 1024 and a temporal resolution of 256. Each point has a geographical location and timestamp and is projected into Web Mercator. This BLOC is then processed with Python scripts to generate the final visualizations. Because the cube is large, we did not attempt to render it in realtime. The Python scripts are used to generate an animation showing the density of points over time, and they generate frames at a rate of two frames per second. An example visualization generated from the BLOC is shown in Figure 6. More examples are shown in the video that accompanies this paper.

Although our implementation is not parallel and runs only on a single node, the BLOC technique is very simple to parallelize because BLOCs are linear. If a dataset is split in half, then the BLOC for the whole dataset is equal to the sum of the BLOCs computed for each half. Assuming that the BLOC size is small enough that it fits in the memory of a single node and can be communicated over the network quickly, BLOC construction is embarrassingly parallel, as each node can independently generate a BLOC for a portion of the dataset, and then all generated BLOCs can be summed together afterwards.

4 Discussion

Computer-based visualizations are defined using discrete programs to process discrete data and display discrete frames. Yet, often this discrete data is a representation of a continuous concept. For example, a floating point number is a discrete approximation of a real number, and real numbers are a continuous set. Similarly, pixels on a screen are both spatially discrete and display a discrete set of colors at discrete moments in time, but the pixel array approximates a continuously-changing spatially continuous image with a continuous color space. Digital audio is also stored as a discrete sequence of discrete samples, but this sequence represents a continuous wave.

Both the pixel approximation to continuous images and the sample approximation to audio rely on limitations of the human perceptual system: both eyes and ears are bandlimited and have bounds on sensitivity. The human ear and eye both have low-pass characteristics where sufficiently high frequencies cannot be perceived. As long as digital audio is sampled at a rate twice that of the highest frequency that the human ear can perceive, a reconstruction should be indistinguishable from the original, provided that each sample is recorded with sufficient resolution that quantization error is below the just noticeable difference threshold. Similarly, if pixels are sufficiently small and have sufficiently high frame-rates and color resolution, they should be perceptually indistinguishable from the continuous case.

However, just as catastrophic cancellation is the bane of numerical computing, aliasing is the bane of sampled and binned representations. Even though a floating point number may have 15 significant figures of precision, if a numerical algorithm isn't carefully designed, most of these significant figures may be wrong in a result. Similarly, if a graphics or audio system is not carefully designed, much of the output bandwidth may be inhabited by aliasing artifacts.

We advocate an approach where visualizations are defined in the continuous domain as a mapping from datasets to continuous images and animations. Only after the visualization algorithm has been defined in the continuous domain should an attempt be made to implement it in a program. By designing first in the continuous domain, a ground truth is established, and the implementation can be checked for accuracy against this ground truth.

We have presented BLOCs, a very general big data visualization technique that provides a discrete representation of a continuous concept. Using the techniques in this paper, BLOCs can be used to construct accurate emulations of continuous-domain visualizations such as kernel density estimation. The key benefit of BLOCs is that they improve performance while retaining the same simple mathematical elegance of continuous-domain techniques.

In practice, BLOCs are useful in many cases where data cubes are useful. Like data cubes, however, BLOCs do not scale well to high-dimensional datasets. However, for cases when it is only necessary to filter based on a few attributes, BLOCs provide a valuable tool for constructing interactive visualizations. Like data cubes, BLOCs can be used as building blocks for a vast array of visualizations. They can be used to show the density of events. Given that mean shift clustering is defined in terms of kernel density estimation, BLOCs can be used to determine the cluster of any location within the BLOC. They can also be used to construct parallel coordinate plots, since the lines drawn in parallel coordinate plots can be seen as spatially-varying projections of a BLOC.

BLOCs could also be used in combination with an existing big data system such as Hadoop, Spark, or Drill. These tools could be used to generate a low-dimensional BLOC for two or three attributes at a time while possibly applying Gaussian filtering on other dimensions. The user could then perform real-time dynamic queries with data cube's attributes and then freeze the dynamic query positions on the current BLOC and request another BLOC with different attributes.

There are many open questions about the design space of BLOCs. For example, some visualization systems that are based on data cube provide a way to visualize the minimum and maximum values along a certain dimension for each bin. Our technique cannot produce these, and it is unclear whether or not there is way to provide this information without introducing aliasing artifacts.

Some visualization systems also provide a way to see average values across bins, e.g. the average receipt amount per month. Our technique can handle this case by storing two separate BLOCs: in the first BLOC, each point is weighted by the receipt amount, and in the second BLOC, all points are weighted equally. To render, kernel plots are calculated for both BLOCs and then each resulting sample from the first plot is divided by the corresponding sample from the second plot. This produces a Kernel Smoothing [22] plot. Kernel density plots can be also be used as inputs to higher-level processes [19], and BLOCs can be used in support of this too.

Future work might explore sparse encoding of BLOCs. The simplest approach to this would be to use hash tables to store all nonzero samples during the initial binning and the subsequent mipmap construction. In this scheme, the server would continue to serve tiles to the client, but these tiles would be sparsely encoded. A more sophisticated approach would transfer unbinned points to the client in areas where the sparse encoding is more space-efficient than the dense encoding. Using this scheme, zooming into sparse regions would not require the server to send new tiles, as the tiles could be generated on-the-fly from the unbinned points on the client. Such an encoding could present a great space savings for high dimensional datasets, and such an encoding would also provide a way for the client to drill-down to the levels of individual points.

5 Related Work

BinX [2] offers a method for visualizing large time series datasets. ImMens [16] provides a method for scalable visualizations based on data cubes. Nano cubes [15] provides an in-memory approach to scalable visualization based on a data structure similar to range trees. All of these techniques exhibit aliasing artifacts.

Multiscale representations similar to ours are very common in signal and image processing. Scale-space representations [24] and pyramid represntations [5] [1], like our technique, compute multiple Gaussian-filtered versions of an input signal using different standard deviations. However, in scale-space and pyramid representations, the goal is to produce a representation that can used as input to computer vision algorithms. In scale-space representations in particular, the Gaussian-filtered signals are then transformed into a tree that captures the extrema of the filtered versions. Wavelets [4] are another multi-scale representation commonly used as input to image processing algorithms.

In computer graphics, mipmaps [23] are used to render images quickly at varying scales. However, the trilinear interpolation described in the paper is less accurate than the scheme we describe. In audio processing, mipmaps are used to quickly resample recorded audio at rates [7]. These mipmaps use brickwall lowpass filters, which are ideal for audio but result in undesirable ringing artifacts in graphical applications. Although a Gaussian filter at the end of the mipmapping could be used to remove most of these artifacts, our technique avoids the introduction of ringing artifacts at all.

Integration-based methods such as summed area tables [6] [13] in computer graphics and integrated wavetable synthesis [10] [9] in audio processing could be adapted for visualization purposes. However, mipmaps likely have better cache performance and allow for tile-based techniques where clients can produce visualizations based on a subset of the whole dataset. Moreover, integration-based methods have problems with numerical stability when used on large datasets.

The simple and commonly-used brute force method to computing kernel density estimation requires execution time proportional to the number of pixels on the screen multiplied by the number of data points for displaying a 2D kernel plot. Even if the kernel could be evaluated with only a single floating point operation (in practice, it would almost certainly require more evaluations), to compute a visualization of size 1024 by 678 for a dataset of 3 billion points at 60 frames per second would require over 283 petaflops of processing power in kernel evaluations alone. Even at the cheap rate of eight cents per gigaflop, this would require an investment of over 22 million dollars per simultaneous user. In practice, the cost would be far greater, since kernel evaluations would almost certainly require more than one floating point operation, and this estimate does not include the processing power required for summing.

Truncating the kernel gives improved performance [14], but with a sufficiently large dataset, it is still prohibitively expensive to perform this technique in real-time. Suppose we require a 10 by 10 kernel, as used in the paper. Then we require 100 kernel evaluations per data point. If each one of these evaluations required a single floating point operation, then rendering at 60 frames per second would require 18 teraflops of processing power, which is beyond the processing power of a single commodity GPU. Moreover, assuming that each data point is represented by two floating point numbers and that all data points need to be traversed at each frame, rendering 60 frames per second would require 11.52 terabits per second of memory bandwidth, which is beyond the capabilities of a single commodity GPU. By contrast, a pre-generated 2D BLOC with mipmapping can used to generate a kernel density plot of any desired within the range satisfiable by the BLOC. Generating the kernel density plot in this case reduces to performing a Gaussian blur on the appropriate section of the BLOC, a task well within reach of a modern GPU.

The fast Gauss transform [11] provides a fast way to compute kernel density estimates at arbitrary locations. However, using this for interactive visualization would still requires a full traversal of the dataset for each frame.

6 Conclusion

We have introduced bandlimited OLAP cubes, an approach to interactive information visualization that uses techniques from digital signal processing to create a condensed version of a large dataset. This condensed version can be used to generate kernel density plots that accurately represent patterns in the source dataset and suppress aliasing artifacts. Moreover, these visualizations can incorporate dynamic query techniques without introducing temporal aliasing artifacts.

We have also introduced a specialized BLOC-based technique that allows for continuous zooming of kernel density plots and continuous brushing. This technique scales readily to large datasets while maintaining interactivity at 60 frames per second.

Acknowledgments

This work was partially supported by NSF grant IIS-1447416 and a grant from L-3 Communications.

References

[1] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. Pyramid methods in image processing. *RCA engineer*, 29(6):33–41, 1984. 2.2, 5

[2] Lior Berry and Tamara Munzner. Binx: Dynamic exploration of time series datasets across aggregation levels. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages p2–p2. IEEE, 2004. 5

[3] PA Bromiley. Products and convolutions of gaussian probability density functions, 2013. 2.1

[4] C Sidney Burrus, Ramesh A Gopinath, and Haitao Guo. Introduction to wavelets and wavelet transforms: a primer. 1997. 5

[5] Peter J Burt. Fast filter transform for image processing. *Computer graphics and image processing*, 16(1):20–51, 1981. 5

[6] Franklin C Crow. Summed-area tables for texture mapping. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 207–212. ACM, 1984. 5

[7] Laurent De Soras. The quest for the perfect resampler, 2003. 5

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 2.3

[9] Andreas Franck and Vesa Välimäki. Higher-order integrated wavetable synthesis. In *Proc. 15th Int. Conf. Digital Audio Effects (DAFx-12), York, UK*, pages 245–252, 2012. 5

[10] Günter Geiger. Table lookup oscillators using generic integrated wavetables. *omega*, 60:50, 2006. 5

[11] Leslie Greengard and John Strain. The fast gauss transform. *SIAM Journal on Scientific and Statistical Computing*, 12(1):79–94, 1991. 5

[12] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *Pervasive Computing, IEEE*, 7(4):12–18, 2008. 3

[13] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. In *Computer Graphics Forum*, volume 24, pages 547–555. Wiley Online Library, 2005. 5

[14] Ove Daae Lampe and Helwig Hauser. Interactive visualization of streaming data with kernel density estimation. In *Pacific Visualization Symposium (PacificVis), 2011 IEEE*, pages 171–178. IEEE, 2011. 5

[15] Lauro Lins, James T Klosowski, and Carlos Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *Visualization and Computer Graphics, IEEE Transactions on*, 19(12):2456–2465, 2013. 5

[16] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. immens: Real-time visual querying of big data. In *Computer Graphics Forum*, volume 32, pages 421–430. Wiley Online Library, 2013. 1, 5

[17] Juhan Nam, Vesa Valimaki, Jonathan S Abel, and Julius O Smith. Efficient antialiasing oscillator algorithms using low-order fractional delay filters. *Audio, Speech, and Language Processing, IEEE Transactions on*, 18(4):773–785, 2010. 2.1, 4

[18] Michael G Paulin. Digital filters for firing rate estimation. *Biological cybernetics*, 66(6):525–531, 1992. 1

[19] Jeff M Phillips, Bei Wang, and Yan Zheng. Geometric inference on kernel density estimates. *arXiv preprint arXiv:1307.7760*, 2013. 4

[20] Julius O. Smith. *Physical Audio Signal Processing*. `http://-ccrma.stanford.edu/~jos/pasp/`, accessed ¡date¿. online book, 2010 edition. 1

[21] Lucas J Van Vliet, Ian T Young, and Piet W Verbeek. Recursive gaussian derivative filters. In *Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on*, volume 1, pages 509–514. IEEE, 1998. 2.2

[22] Matt P Wand and M Chris Jones. *Kernel smoothing*, volume 60. Crc Press, 1994. 4

[23] Lance Williams. Pyramidal parametrics. In *Acm siggraph computer graphics*, volume 17, pages 1–11. ACM, 1983. 5

[24] Andrew P Witkin. Scale-space filtering: A new approach to multi-scale description. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'84.*, volume 9, pages 150–153. IEEE, 1984. 5