Hierarchical Bin Buffering: Online Local Moments for Dynamic External Memory Arrays

DANIEL LEMIRE
Université du Québec à Montréal
and
OWEN KASER
University of New Brunswick

For a massive I/O array of size n, we want to compute the first N local moments, for some constant N. Our simpler algorithms partition the array into consecutive ranges called bins, and apply not only to local-moment queries, but also to algebraic queries. With N buffers of size $\sqrt{n}$, time complexity drops to $O(\sqrt{n})$. A more sophisticated approach uses hierarchical buffering and has a logarithmic time complexity ($O(b\log_b n)$), when using N hierarchical buffers of size n/b. Using Overlapped Bin Buffering, we show that only one buffer is needed, as with wavelet-based algorithms, but using much less storage.

Categories and Subject Descriptors: H.3.5 [**Information Storage**]: Online Information Services; G.1.1 [**Numerical Analysis**]: Interpolation

General Terms: Algorithms, Theory, Experimentation

Additional Key Words and Phrases: Very Large Arrays, Hierarchical Buffers, Statistical Queries, Polynomial Fitting

1. INTRODUCTION

In a data-driven world where permanent storage devices have an ever growing capacity, I/O access becomes a bottleneck when read/write performance does not increase as quickly as capacity [Vitter 2002]. As Gray put it: "we're able to store more than we can access" [Patterson 2003]. At the same time, users of applications in OLAP [Codd et al. 1993] or in visualization [Silva et al. 2002] now expect online processing of their data sets. A strategy to solve this problem is to use a relatively small internal-memory buffer to precompute some of the expected queries. For example, it might be reasonable to use a memory buffer of one megabyte (1 MB)[1] per terabyte (TB) of external-memory data.

If X is a random variable with probability distribution f, we define the moment of order N about a value c as the expectation of $(X-c)^N$ or $E((X-c)^N) = \int (x-c)^N f(x)dx$. Correspondingly, given an array a and some constant c, $\sum_i (i-c)^N a_i$ is a moment of order N. Given an arbitrarily large array a, we can precompute the moments with ease, however we are interested in **local** moments: given a range of indices $p,\ldots,q$ and a constant c all pro-

[1]Throughout, we use the familiar units of kB, MB, GB and TB to measure storage in groups of 2^{10}, 2^{20}, 2^{30} and 2^{40} bytes. Our units thus coincide with the new IEC units KiB, MiB, GiB and TiB [IEC 1999].

The first author was supported by NSERC grant 261437 and the second author was supported by NSERC grant 155967.
Authors' addresses: Daniel Lemire, Université du Québec à Montréal, 100 Sherbrooke West, Montréal, QC H2X 3P2 Canada and Owen Kaser, University of New Brunswick, 100 Tucker Park Road, Saint John, NB E2L 4L1 Canada

vided dynamically, we wish to compute $\sum_{i=p}^{q}(i-c)^N a_i$ online. Frequency moments [Alon et al. 1996], $\sum_i f_i^k$ where f_i is the number of occurences of item i, are outside the scope of this paper.

Local moments are used widely, from pattern recognition and image processing to multidimensional databases (OLAP). Among other things, they have been proposed as a replacement for maximum likelihood methods in statistics, to cope with the magnitude of the new data sets [Scott and Sagae 1997].

As an example application, given the number of items in a store for every possible price, the sum (moment 0) over a range of prices would return how many items are available, the first moment would return the total dollar value for items in the specified price range, and the second moment would allow us to compute the standard deviation and variance of the price.

As another example, imagine a moving sensor measuring the density of some metallic compound in the ground. A geophysicist could then ask for the average density of this compound over some terrain or for the "center of mass" in some region. Indeed, if the array a contains the density over some one-dimensional strip of terrain, then the average density over some region given by indices $p,\ldots,q$ is given by $\sum_{i=p}^{q} a_i/(q-p)$ whereas the center of mass is given by $(\sum_{i=p}^{q} ia_i)/\sum_{i=p}^{q} a_i$.

As yet another example, consider the local regression problem [Cleveland and Loader 1995]. Suppose that given a large segment $p,\ldots,q$ of a very large array y_i, the user wants to view the best polynomial of order $N-1$ fitting the data: this problem occurs when trying to segment time series [Lemire 2007], for example. Given a polynomial $\sum_{k=0}^{N-1} a_k x^k$, the residual energy is given by $\sum_{i=p}^{q}(y_i - \sum_{k=0}^{N-1} a_k i^k)^2$. Setting the derivative with respect to a_l to zero for $l = 0,\ldots,N-1$, we get a system of N equations in N unknowns, $\sum_{k=0}^{N-1} a_k \sum_{i=p}^{q} i^{k+l} = \sum_{i=p}^{q} y_i i^l$ where $l = 0,\ldots,N-1$. Note that the right-hand-sides of the equations are N local moments whereas on the left-hand-side, we have the $N \times N$ matrix $A_{k,l} = \sum_{i=p}^{q} i^{k+l}$. For any $k+l$, we can compute the sum $\sum_{i=p}^{q} i^{k+l}$ in constant time irrespective of $q-p$. To see this, consider that because $\sum_{i=p}^{q} i^{k+l} = \sum_{i=0}^{q} i^{k+l} - \sum_{i=0}^{p-1} i^{k+l}$, it is sufficient to be able to compute expressions of the form $\sum_{i=0}^{q} i^{k+l}$ quickly. However, there is a formula for these summations.

Several fast techniques have been proposed to compute local moments [Li and Shen 1992; Zhou and Kornerup 1995] but this paper is concerned with **precomputing** auxiliary information to speed up the computation of local moments.

2. NOTATION

We use C-like indexing for arrays: an array of length n is indexed from 0 to $n-1$ as in $a_0,\ldots,a_{n-1}$. Indices are always integers, so that the notation $i \in [k,l]$ for an index means that $i \in \{k,\ldots,l\}$. Given a set R, the set of all finite arrays of values in R is denoted by $\mathcal{A}^R$. The restriction of a function f to domain D is noted $f_{|D}$. Some common functions over real-valued arrays ($\mathcal{A}^{\mathbb{R}}$) or "range-query functions" include COUNT and SUM, where $\text{COUNT}(a_0,\ldots,a_{n-1}) = n$ and $\text{SUM}(a_0,\ldots,a_{n-1}) = \sum_{i=0}^{n} a_i$. Other possible queries include MAX (which returns the maximum value in a range) or MAX_N (which returns the N largest values found in a range). The query moment of order N is formally defined as $\sum_{i=p}^{q}(i-p)^N a_i$. Note that computing query moments of order N, for various N, allows the

calculation of any local moment. Indeed, notice that

$$\sum_{i=p}^{q}(i-c)^N a_i = \sum_{k=0}^{N}\binom{N}{k}(p-c)^{N-k}\sum_{i=p}^{q}(i-p)^k a_i \tag{1}$$

for any constant c, by the expansion of $((i-p)+(p-c))^N$ using the Binomial Theorem.

3. RELATED WORK

Using a buffer, we can compute linear range queries in $O(1)$ time. For example, given an array $A = \{a_0, a_1, \ldots, a_{n-1}\}$, we can use the Prefix Sum method [Ho et al. 1996] and precompute the array $PS(A) = \{a_0, a_0+a_1, a_0+a_1+a_2, \ldots, a_0+\cdots+a_{n-1}\}$. By a mere subtraction, we can then compute any range sum of the form $a_k + a_{k+1} + \cdots + a_l$ since

$$a_k + \cdots + a_l = (a_0 + \cdots + a_l) - (a_0 + \cdots + a_{k-1}).$$

However, if a few data points are updated, then all of $PS(A)$ may need to be recomputed: updates require $O(n)$ time. A more robust approach is the Relative Prefix Sum (RPS) method [Geffner et al. 1999] which buffers the prefix sums only locally over blocks of size b. For example,

$$RPS(A) = \{a_0, a_0+a_1, a_0+a_1+a_2, a_3, a_3+a_4, \ldots\}$$

when $b = 3$. Clearly, $RPS(A)$ can be updated in time $O(b)$. To still achieve $O(1)$ query time, we use an *overlay* buffer

$$a_0+a_1+a_2, a_0+\cdots+a_5, \ldots, a_0+\cdots+a_{n-1}$$

that can be updated in time $O(n/b)$. While RPS requires $\Theta(n)$ in storage, updates can be done in $O(\sqrt{n})$ time by choosing $b = \sqrt{n}$. We can improve the update performance of RPS using the Pyramidal Prefix Sum (PyRPS) method [Lemire 2002]. Its query complexity is $O(\rho)$ with update cost $O(\rho n^{1/\rho})$, where $\rho = 2, 3, \ldots$ Thus, PyRPS obtains constant-time queries but with faster updates than RPS. Furthermore, PyRPS supports queries and updates in logarithmic time by choosing $\rho = \log(n)$.

Of course, these techniques extend to other range queries. In the context of orthogonal range queries for multidimensional databases, similar results are even possible for range-maximum queries [Poon 2003].

The PS, RPS, and PyRPS methods (and similar alternatives for other queries) have a common inconvenience: each type of range query is buffered separately: $\sum_k a_k$, $\sum_k k a_k$, $\sum_k k^2 a_k$, … An equivalent view is of a single *tuple-valued* buffer, rather than several real-valued buffers. The ProPolyne framework [Schmidt and Shahabi 2002] showed how to avoid tuple-valued buffers by using wavelets. ProPolyne has both logarithmic queries and updates at the cost of a buffer of size $O(n)$ for computing local moments (*Polynomial Range Queries*). ProPolyne simultaneously buffers all local moments up to a given degree. So, ProPolyne reduces storage when compared with prefix-sum methods such as PyRPS, but at the expense of constant-time queries. See Table I for a comparison of various alternatives to buffer local moments.

In a wavelet framework such as ProPolyne, we can keep only the most significant wavelet coefficients to reduce storage and increase performance, while obtaining reasonably accurate results [Chakrabarti et al. 2001; Jahangiri et al. 2005; Vitter et al. 1998]. It is also possible to process the queries incrementally so that approximate results are available sooner.

Table I. Comparison of local moment algorithms with corresponding storage requirements and complexity for large n where we buffer the first N moments. Note that $\rho = 2, 3\ldots$ is a parameter that can be chosen to be large. The storage requirement is the number of components needed to buffer computations. OLA is a form of Bin Buffering specific to local moments.

Algorithm	Query	Update	Storage
ONE-SCALE OLA	$O(Nn/b+N^2b)$	$O(N)$	$n/b+1$
HIERARCHICAL OLA	$O(N^2b\log_b n)$	$O(N^2\log_b n)$	$n/b+1$
Bin Buffering	$O(Nn/b+Nb)$	$O(N)$	Nn/b
Hierarchical Bin Buffering	$O(Nb\log_b n)$	$O(N\log_b n)$	Nn/b
ProPolyne	$O(N^2\log_2 n)$	$O(N^2\log_2 n)$	n
Prefix Sums	$O(N)$	$O(Nn)$	Nn
Relative Prefix	$O(N)$	$O(N\sqrt{n})$	Nn
PyRPS	$O(N\rho)$	$O(N\rho\sqrt[\rho]{n})$	Nn
PyRPS (log)	$O(N\log n)$	$O(N\log n)$	Nn

4. CONTRIBUTION AND ORGANIZATION

For storage, a reduction from n to $\sqrt{n}$ can be quite significant: if $n = 2^{40}$ (1 TB), then $\sqrt{n} = 2^{20}$ (1 MB), so we argue that simple buffering schemes might often prove more practical than PyRPS or ProPolyne, especially because a small buffer can be generally constructed faster. As Ho et al. [Ho et al. 1996] observed (for the Prefix Sum Method), good performance can be obtained with a small buffer, provided we retain access to the original array.

The paper is organized as follows. We first consider how bin buffers can be used to speed up many range queries over dynamic external arrays (section 5). For each bin or "range of indices," Bin Buffering [Moerkotte 1998] associates a single buffer component. However, its scalability is limited because very large buffers do not improve performance and can even worsen it. In section 6, we present the analysis of a variant, Hierarchical Bin Buffering, which supports logarithmic queries and updates even for modest buffers. These results are novel, but have been alluded to in the concluding section of a paper [Moerkotte 1998]. In section 7, we present a novel buffering framework: Overlapped Bin Buffering. In Overlapped Bin Buffering, each buffer component depends not only on one but a range of bins. In this context, we present Lagrange Interpolation (section 8) as a tool to compute a useful buffer and establish an explicit link between buffering and interpolation. The result is an OLA buffer and we show it can be hierarchical as well. Section 9 presents precisely stated algorithms and experimental results. We then proceed on some concluding remarks. We elaborate on the differences between OLA and BIN BUFFERING (section 10) and show how OLA can support efficient progressive approximate queries (section 11).

5. FAST ALGEBRAIC RANGE QUERIES USING PRECOMPUTATION OVER BINS

Let R be an algebraic structure such as $\mathbb{R}$ or $\mathbb{R}^m$. A range-query function $Q : \mathcal{A}^R \to R$ is *distributive* [Gray et al. 1996] if there is a function $F : \mathcal{A}^R \to R$ such that for all $0 \leq k < n-1$,

$$Q(a_0,\ldots,a_k,a_{k+1},\ldots,a_{n-1}) = F(Q(a_0,\ldots,a_k),Q(a_{k+1},\ldots,a_{n-1})).$$

Examples of distributive range-query functions include COUNT, SUM, and MAX. In this paper, we shall only consider range queries where the computational cost is indepen-

dent of the values being aggregated. By convention, $F(Q(a_0,\ldots,a_k)) = Q(a_0,\ldots,a_k)$; i.e., the function F is the identity when applied to a single value.

We have

$$\begin{aligned} Q(a_0,\ldots,a_{n-1}) &= F(Q(a_0,\ldots,a_{n-1})) \\ &= F(Q(a_0,\ldots,a_{n-2}),Q(a_{n-1})) \\ &= F(F(Q(a_0,\ldots,a_{n-2})),Q(a_{n-1})) \\ &= F(F(F(Q(a_0,\ldots,a_{n-3}),Q(a_{n-2})), \\ &\quad Q(a_{n-1})) \\ &= \cdots \end{aligned}$$

and so we can compute $F(Q(a_0,\ldots,a_{n-1}))$ recursively, using $n-1$ pairwise aggregations. Hence, a distributive range query over n terms has complexity $O(n)$.

Distributive range-query functions can be combined. For example, define the joint query function (Q_1,Q_2) as the tuple-valued query function

$$(Q_1,Q_2)(a_0,\ldots,a_{n-1}) = (Q_1(a_0,\ldots,a_{n-1}),Q_2(a_0,\ldots,a_{n-1})).$$

We can verify that (Q_1,Q_2) is distributive if Q_1 and Q_2 are distributive.

In this paper, a real-valued range-query function $Q:\mathcal{A}^{\mathbb{R}} \to \mathbb{R}$ is *algebraic* if there is an intermediate tuple-valued **distributive** range-query function $G:\mathcal{A}^{\mathbb{R}} \to \mathbb{R}^m$ from which Q can be computed. For example, given the tuple (COUNT, SUM), one can compute AVERAGE by a mere ratio, so AVERAGE is an example of an algebraic query function. In other words, if Q is an algebraic function then there must exist G and $F:\mathcal{A}^{\mathbb{R}^m} \to \mathbb{R}^m$ for some fixed integer m such that

$$G(a_0,\ldots,a_k,a_{k+1},\ldots,a_{n-1}) = F(G(a_0,\ldots,a_k),G(a_{k+1},\ldots,a_{n-1})).$$

We have that algebraic queries can be computed in time $O(n)$ because we can interpret them as distributive queries over tuples. All real-valued distributive range-query functions are algebraic, but examples of non-distributive algebraic functions include MAX_N, AVERAGE, CENTER_OF_MASS, STANDARD_DEVIATION, and local moments. As an example, if Q is AVERAGE, then we can choose G to compute the tuples (COUNT, SUM) and F can be the component-wise sum. Similarly, if Q is the local moment of order N, then G should compute the tuple made of (COUNT, SUM, …, moment of order N). Combining two N^{th}-order query moments $\sum_{i=p}^{r-1}(i-p)^N a_i$ and $\sum_{i=r}^{q}(i-r)^N a_i$ into an aggregate $\sum_{i=p}^{q}(i-p)^N a_i$ can be done by Eq. (1), which requires that we know the lower-order moments. In terms of N, a straightforward implementation of G runs in O(N) time. Indeed, we can compute $G(a_0,\ldots,a_k)$ as $F(G(a_0,\ldots,a_{k-1}),G(a_k))$ but $G(a_k) = (1,a_k,0,\ldots,0)$. Operation F itself requires $O(N)$ time. Hence, $G(a_0,\ldots,a_k)$ can be computed in $O(Nn)$ time.

In what follows, we will specify algebraic functions as triples (Q,G,F). Whenever we use the (Q,G,F) notation, it is understood that

$$Q:\mathcal{A}^{\mathbb{R}} \to \mathbb{R}, \text{ and } G:\mathcal{A}^{\mathbb{R}} \to \mathbb{R}^m, \text{ and } F:\mathcal{A}^{\mathbb{R}^m} \to \mathbb{R}^m$$

and often Q will not be used explicitly because computing G is enough. We will assume that the size of the tuples, m, is small: $m \lesssim 16$.

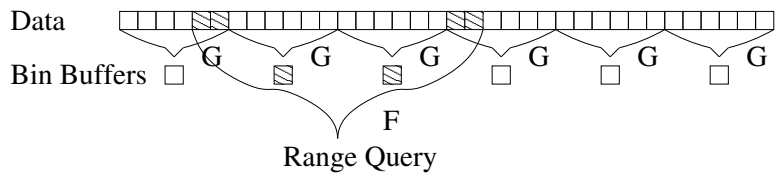

Fig. 1. An algebraic range query supported by Bin Buffering, as in Eq. (2).

Given an integer b that divides n and given an algebraic function (Q,G,F), we can buffer queries by precomputing b/n components

$$\begin{aligned}&G(a_0,\ldots,a_{b-1}),\\&G(a_b,\ldots,a_{2b-1}),\ldots,\\&G(a_{n-b},\ldots,a_{n-1})\end{aligned}$$

denoted $B_0,\ldots,B_{n/b-1}$. This buffer can be updated in time $O(b)$ if an array component is changed. Using this precomputed array, range queries can be computed in time $O(n/b+b)$ because of the formula

$$G(a_k,\ldots,a_l) = F(G(a_k,\ldots,a_{b\lceil k/b\rceil-1}),B_{\lceil k/b\rceil},\ldots,B_{\lfloor l/b\rfloor-1},G(a_{b\lfloor l/b\rfloor},\ldots,a_l)). \quad (2)$$

See also Figure 1. By choosing $b=\sqrt{n}$, we get updates and queries in time $O(\sqrt{n})$ with a buffer of size $\sqrt{n}$. In different terms, this algorithm was presented by Moerkotte [Moerkotte 1998].

When buffering local moments of order N, G computes $N+1$-tuples so that the size of the buffer is $(N+1)\times n/b$. This can be reduced to $N\times n/b$ if all bins are of a fixed size b, since we need not store COUNT.

An algebraic range-query function (Q,G,F) is *linear* if the corresponding intermediate query G satisfies

$$G(a_0+\alpha d_0,\ldots,a_{n-1}+\alpha d_{n-1}) = G(a_0,\ldots,a_{n-1})+\alpha G(d_0,\ldots,d_{n-1})$$

for all arrays a,d, and constants α. SUM, AVERAGE and local moments are linear functions; MAX is not linear. Linear queries over bins of size b can be computed using the formula $G(a_0,\ldots,a_{b-1}) = a_0G(e^{(0)})+\ldots+a_{b-1}G(e^{(b-1)})$, where $e^{(i)}$ an array of size b satisfying $e^{(i)}_j=0$ if $i\neq j$ and $e^{(i)}_i=1$. For our purposes, we define an update by the location of the change, k, and by how much the value changed, $\Delta = a'_k - a_k$. We see that the update complexity for buffered linear range queries is reduced to constant time since

$$\begin{aligned}G(a_0,\ldots,a_{k-1},a'_k,a_{k+1},\ldots,a_{b-1}) - G(a_0,\ldots,a_k,\ldots,a_{b-1}) &= (a'_k-a_k)G(e^{(k)})\\&= G(e^{(k)})\Delta\end{aligned}$$

and $G(e^{(k)})$ can be precomputed or computed in constant time.

Hence, we see that:

(1) All algebraic queries can be bin buffered, including MAX, AVERAGE, and local moments.

(2) For linear queries, the buffer can be updated quickly.

LEMMA 5.1. *For an algebraic range-query function Q, given an array of size n, Bin Buffering uses a buffer of n/b tuples computed in time $O(n)$ and updated in time $O(b)$ to support queries in time $O(n/b+b)$. Choosing $b=\sqrt{n}$ minimizes the query complexity to $O(\sqrt{n})$. If Q is linear then updates take constant time.*

LEMMA 5.2. *Consider local moments of degree N, where N is fixed and small. Given an array of size n, Bin Buffering uses a buffer of size $N \times n/b$ computed in time $O(n)$ and updated in constant time to support queries in time $O(n/b+b)$. Choosing $b=\sqrt{n}$ minimizes the query complexity to $O(\sqrt{n})$.*

If N is not considered fixed, buffer computation is $O(Nn)$, update time is $O(N)$ and query time is $O(Nn/b+bN)$.

A possible drawback of the Bin Buffering algorithm is that the query complexity cannot be reduced by using a larger buffer. For example, in going from a buffer of size $n/2$ to a buffer of size $\sqrt{n}$, the algorithm's complexity goes down from $O(n)$ to $O(\sqrt{n})$. We will show in the next section how we can use larger buffers in a hierarchical setting to increase the speed.

6. HIERARCHICAL BIN BUFFERING

In the previous section, we showed we could precompute algebraic range queries over bins of size b to support $O(n/b+b)$-time queries. We can scale this up using a pyramidal or hierarchical approach [Lemire 2002].

For a fixed b, the n/b term dominates the $O(n/b+b)$ complexity. In Eq. (2) the n/b term comes from the buffer aggregation. So, we started from an aggregation over n terms and reduced it to an aggregation over n/b terms; clearly we can further reduce the aggregation over n/b terms to an aggregation over n/b^2 terms by the same technique (see Figure 2). In other words, we can buffer the buffer. Hence, considering the buffer of size n/b as a source array, we can buffer it using n/b^2 components to support queries in time $O(n/b^2+b)$ over the buffer instead of $O(n/b)$. Thus, the end result is to have queries in time $O(n/b^2+2b)$ with a buffer of size at most $n/b+n/b^2$. Repeating this argument $\log_b n$ times, we get queries in time $O(b\log_b n)$ using $\sum_{k=1,\ldots,\log_b n} n/b^k \leq n/(b-1)$ storage. If the query function is *invertible*, as defined in the next subsection, then we can use in-place storage for higher-scale buffers. This reduces the internal memory usage to n/b. The update complexity is $O(b\log_b n)$ in general and $O(\log_b n)$ for linear queries.

6.1 In-place Storage for Invertible Query Functions

A given algebraic function (Q,G,F), is *invertible* if O(1) time is sufficient to solve for x in $z=F(x,y)$, where $x,y,z \in \mathbb{R}^m$ (m is assumed small). Linear queries are invertible, and being invertible is a useful property: it means that the storage used by x can be used to store z — storing x,y or z,y is almost equivalent. This lets us "buffer a buffer" in place, as the next proposition shows.

PROPOSITION 6.1. *If (Q,G,F) is an invertible algebraic query function, then second-scale Bin Buffer components $B'_0=F(B_0,\ldots,B_{b-1})$ and $B'_b=F(B_b,\ldots,B_{2b-1}),\ldots$ can be stored in-place at positions $0,b,2b,\ldots$ in the buffer (overwriting values $B_0,B_b,B_{2b},\ldots$) without increasing query time complexity.*

PROOF. Assume we use in-place storage for the second-scale Bin Buffers $B'_0,B'_1,\ldots$, overwriting $B_0,B_b,\ldots$. We must evaluate expressions of the form $F(B_k,\ldots,B_l)$, which can

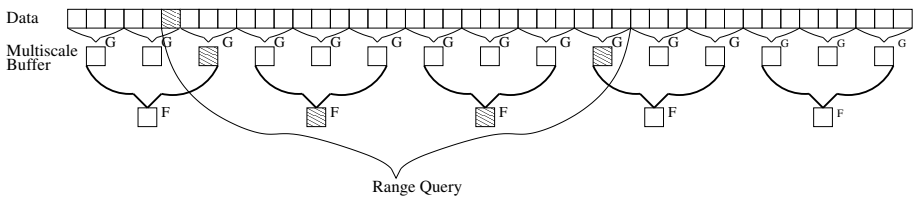

Fig. 2. An algebraic range query supported by Hierarchical Bin Buffering. We essentially repeat Figure 1 over the buffer itself. In the example given, by aggregating buffer components, we replace 6 first-scale buffer components by 2 second-scale components.

be done using the second-scale Bin Buffer, according to the formula

$$F(B_k,\ldots,B_l) = F(F(B_k,\ldots,B_{\lceil k/b\rceil b-1}),B'_{b\lceil k/b\rceil},\ldots,B'_{b(\lfloor l/b\rfloor-1)},F(B_{b\lfloor l/b\rfloor},\ldots,B_l)).$$

The only place where an overwritten value appears is in the last term: $B_{b\lfloor l/b\rfloor}$ has been replaced by the value of $B'_{b\lfloor l/b\rfloor}$. However, the query is invertible, so $B_{b\lfloor l/b\rfloor}$ can be recovered in constant time. Thus the algorithm using two-scale buffers is still going to be $O(n/b^2+2b)$, even though in-place storage has been used. □

We can repeat this process for each buffer scale, each time incurring only a fixed cost for recovering an overwritten value. The total additional cost is $O(\log_b n)$, but this is dominated by the cost of the query itself ($O(b\log_b n)$). In other words, in-place storage almost comes for free.

As an example of Hierarchical Bin Buffering with in-place storage, consider the array $a_0,\ldots,a_{80}$ ($n=81$) and some invertible algebraic query function (Q,G,F). The one-scale Bin Buffering algorithm with $b=3$ simply precomputes

$$B_0 = G(a_0,a_1,a_2), B_1 = G(a_3,a_4,a_5),\ldots,B_{26} = G(a_{78},a_{79},a_{80})$$

so that if we want $Q(a_1,\ldots,a_{79})$, we still have to compute

$$F(G(a_1,a_2),B_1,\ldots,B_{25},G(a_{78},a_{79}))$$

which is the aggregation of 27 terms using F. We can aggregate the buffer itself in a second buffer, in this case, by precomputing

$$B'_0 = F(B_0,B_1,B_2), B'_3 = F(B_3,B_4,B_5),\ldots,B'_{24} = F(B_6,B_7,B_8)$$

and storing them in-place, so that $B_0,B_3,\ldots,B_{24}$ are replaced by the newly computed $B'_0,B'_3,\ldots,B'_{24}$. Then, to compute $Q(a_1,\ldots,a_{79})$, it suffices to compute

$$F(G(a_1,a_2),B_1,B_2,B'_3,B'_6,\ldots,B'_{21},B_{24},B_{25},G(a_{78},a_{79})),$$

the aggregation of only 13 terms. The query cost is halved without using any additional memory.

As the next lemma explains, the hierarchical case presented above is simply a generalization of the case in the previous section, but where large buffers can be used to answer queries in logarithmic time. Recall that local moments are linear and invertible whereas MAX queries are neither.

LEMMA 6.2. *For an algebraic range-query function Q, given an array of size n, Hierarchical Bin Buffering uses a buffer of* $\frac{n}{b-1}$ *tuples computed in time* $O(n)$ *and updated in time* $O(b\log_b n)$ *to support queries in logarithmic time* $O(b\log_b n)$. *If the query is invertible, then a smaller memory buffer of size* n/b *can be used; for linear queries updates can be done in time* $O(\log_b n)$.

For non-invertible queries such as MAX, that is, the worst case scenario, this last lemma implies that a buffer of size $n/(b-1)$ can support queries in time $O(b\log_b n)$ with updates in time $O(b\log_b n)$. For invertible and linear queries such as SUM, the storage is only n/b with updates in time $O(\log_b n)$. Choosing $b = 2$ minimizes the query complexity ($O(\log_2 n)$) while maximizing the storage requirement at $n/2$, whereas choosing $b = \sqrt{n}$ reduces to the non-hierarchical (one-scale) case with a query complexity of $O(\sqrt{n})$ and a storage requirement of $\sqrt{n}$.

Note that G operates on tuples, and thus a buffer of n/b elements occupies mn/b space, offsetting the economical nature of the Hierarchical Bin Buffering algorithm. Another result is that the G operation becomes more expensive for higher-order moments; in the analysis leading up to Lemma 6.2, we implicitly assumed the cost of G was constant. However, if the analysis considers that operation costs increase with N, we have that buffer construction is in $O(Nn)$, updates are in $O(N\log_b n)$ and queries are in $O(Nb\log_b n)$. As we shall see in the next section, for some types of queries such as local moments, it is possible to avoid using tuples in the buffer.

7. OVERLAPPED BIN BUFFERING

In the previous sections, we described Bin Buffering and Hierarchical Bin Buffering as it applies to all algebraic queries. Such Bin Buffering is characterized by the facts that buffer components, $B_0 = G(a_0,\ldots,a_{b-1})$, $B_1 = G(a_b,\ldots,a_{2b-1})$, ..., are over disjoint bins and are aggregated using G itself. In this section, we will consider only weighted sums as aggregate operators, and we will define buffer components that depend on several bins at once. This can also be interpreted as having overlapping bins. Our motivation is to buffer local moments using a single real-valued buffer, and we begin by considering one-scale buffering in subsections 7.1–8.1, but in subsection 8.2 we will extend our results to the hierarchical case.

7.1 General Case

Consider an array a of size n indexed as $a_0,\ldots,a_{n-1}$. By convention, $a_j = 0$ for $j \notin [0,n)$. Assuming that n is divisible by b, we group the terms in bins of size b: first $a_0,\ldots,a_{b-1}$, then $a_b,\ldots,a_{2b-1}$ and so on. We have n/b bins and we want to compute $n/b+1$ buffer components $B_0,\ldots,B_{n/b}$ to speed up some range-query functions such as SUM. However, we drop the requirement that each buffer component correspond to one and only one bin, but rather, we allow buffer components to depend on several bins, hence the term "Overlapped Bin Buffering."

For an array $a_0,\ldots,a_{n-1}$ and given integers $M,M' \geq 0$, consider buffer components of the form

$$B_k = \sum_{j=-Mb}^{M'b-1} c_j a_{j+kb}$$

where the coefficients c_j are to be determined but are zero outside their range, i.e., $c_j = 0$

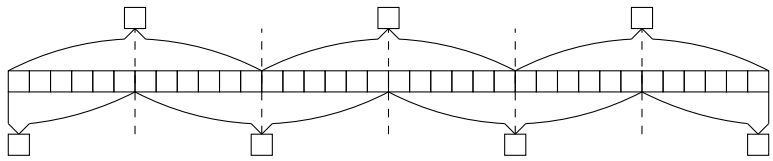

Fig. 3. Overlapped Bin Buffering with $M = 1, M' = 1$. Buffered components depend on overlapping areas.

if $j \notin [-Mb, M'b)$. We could generalize this framework so that $M + M'$ remains a constant but that M and M' depend on the bins: we could accommodate the end and the beginning of the array so that $j + kb$ is always in $[0, n)$, but this makes the formulas and algorithms more pedantic. In essence, the B_k are weighted sums over a range of $M + M'$ bins. We can interpret the coefficients c_j as weights used to compute the buffer component B_k, where j gives the offset in the original array with respect to kb. An example is shown in Figure 3. Note that in the special case where $M = 0, M' = 1$, we have the usual Bin Buffering approach, which maps each bin to exactly one buffer component and where $c_j = 1$ for $j \in [0, b)$. As we shall see, increasing M and M' allows for additional degrees of freedom.

We can compute $\sum_{i=p}^{q} f(i)a_i$, by replacing f by $\tilde{f}$ such that $\tilde{f}$ agrees with f on $[p, q]$ but is zero otherwise, and then compute $\sum_{i=0}^{n-1} \tilde{f}(i)a_i$. Thus, it is enough to have a fast algorithm to compute $\sum_{i=0}^{n-1} f(i)a_i$ for an arbitrary f. The next proposition presents a formula that is instrumental in achieving a fast algorithm.

PROPOSITION 7.1. *Given an array $a_0, \ldots, a_{n-1}$ and integers $M, M' \geq 0$, and given the buffer components $B_k = \sum_{j=-Mb}^{M'b-1} c_j a_{j+kb}$, we have*

$$\sum_{i=0}^{n-1} f(i)a_i = \sum_{k=0}^{n/b} f(kb)B_k + \sum_{i=0}^{n-1} \delta(i)a_i$$

where

$$\delta(i) = f(i) - \sum_{k=\lfloor \frac{i}{b} \rfloor - M' + 1}^{\lfloor \frac{i}{b} \rfloor + M} f(kb)c_{i-kb}.$$

PROOF. By the definition of B_k, we have

$$\sum_{k=0}^{n/b} f(kb)B_k = \sum_{k=0}^{n/b} \sum_{j=-Mb}^{M'b-1} f(kb)c_j a_{j+kb}.$$

Define $i = j + kb$ so that

$$\sum_{k=0}^{n/b} \sum_{j=-Mb}^{M'b-1} f(kb)c_j a_{j+kb} = \sum_{k=0}^{n/b} \sum_{i=kb-Mb}^{kb+M'b-1} f(kb)c_{i-kb}a_i.$$

Because c_{i-kb} is zero whenever $i \notin [kb - Mb, kb + M'b)$, we can replace $\sum_{i=kb-Mb}^{kb+M'b-1}$ by $\sum_{i=0}^{n-1}$ in the above equation to get, after permuting the sums,

$$\sum_{k=0}^{n/b} f(kb)B_k = \sum_{i=0}^{n-1} \left(\sum_{k=0}^{n/b} f(kb)c_{i-kb} \right) a_i.$$

However, note that c_{i-kb} is zero whenever $kb \notin (i-M'b, i+Mb]$ or $k \notin (\lfloor \frac{i}{b} \rfloor - M', \lfloor \frac{i}{b} \rfloor + M]$. Hence, we can replace $\sum_{k=0}^{n/b}$ by $\sum_{k=\lfloor \frac{i}{b} \rfloor - M'+1}^{\lfloor \frac{i}{b} \rfloor + M}$ to get

$$\sum_{k=0}^{n/b} f(kb)B_k = \sum_{i=0}^{n-1} \left(\sum_{k=\lfloor \frac{i}{b} \rfloor - M'+1}^{\lfloor \frac{i}{b} \rfloor + M} f(kb)c_{i-kb} \right) a_i,$$

which can be subtracted from $\sum_{i=0}^{n-1} f(i)a_i$ to prove the result. □

The key idea is that to support fast computations, we want

$$\delta(i) = f(i) - \sum_{k=\lfloor \frac{i}{b} \rfloor - M'+1}^{\lfloor \frac{i}{b} \rfloor + M} f(kb)c_{i-kb} = 0 \tag{3}$$

for most integers i, as this implies that we can compute almost all of the range query using only the buffer: i.e., from the previous proposition when $\delta(i) = 0$, we have

$$\sum_{i=0}^{n-1} f(i)a_i = \sum_{k=0}^{n/b-1} f(kb)B_k.$$

It seems remarkable that the precomputed B_k values are suitable for use with many functions, possibly including functions not envisioned when the buffer was initially constructed.

From $\delta(i) = 0$ we will arrive at Lagrange interpolation in section 8, since we are mostly interested in the case where f is locally a polynomial. Moreover, because we want the ability to store the buffer component B_k at position kb, we also require that $\delta(kb) = 0$. This will ensure that the value a_{kb} is never needed in Eq. (3).

PROPOSITION 7.2. *From the definition*

$$\delta(i) = f(i) - \sum_{k=\lfloor \frac{i}{b} \rfloor - M'+1}^{\lfloor \frac{i}{b} \rfloor + M} f(kb)c_{i-kb},$$

we have that $\delta(kb) = 0$ *for all integers* k *if*

$$c_{lb} = \begin{cases} 1 & \text{if } l = 0 \\ 0 & \text{otherwise} \end{cases}.$$

PROOF. Assume that c_{lb} is zero whenever $l \neq 0$, then

$$\begin{aligned} \delta(lb) &= f(lb) - \sum_{k=l-M'+1}^{l+M} f(kb)c_{(l-k)b} \\ &= f(lb) - f(lb)c_0 \end{aligned}$$

and the result follows. □

Consider the case where $M = 0, M' = 1$, then $B_k = \sum_{j=0}^{b-1} c_j a_{j+kb}$. Suppose we want to buffer range sums, then f will be 1 except at the endpoints. So, from $\delta(i) = 0$, we see that we want

$$\sum_{k=\lfloor \frac{i}{b} \rfloor}^{\lfloor \frac{i}{b} \rfloor} c_{i-kb} = 1$$

or $c_{i-\lfloor \frac{i}{b} \rfloor b} = 1$; that is, c is always 1 within the range of its indices. Thus, we retrieve the formula $B_k = \sum_{j=0}^{b-1} a_{j+kb}$ as the unique solution to buffer range sums when $M = 0, M' = 1$. As we shall see, for larger overlaps the solution is no longer unique and the problem becomes more interesting.

7.2 An Example: Sum and First Moments ($M = 1, M' = 1$)

With $M = 0, M' = 1$, we can buffer SUM queries. Using $M = 1, M' = 1$, we will buffer the first two local moments: local range sums ($\sum_i a_i$) and local first moments ($\sum_i i a_i$). While we will support a wider range of queries, the buffer size remains unchanged. However, the complexity of the queries does go up when N increases.

With $M = 1, M' = 1$, $\delta(i) = 0$ implies

$$\delta(i) = f(i) - \sum_{k=\lfloor \frac{i}{b} \rfloor}^{\lfloor \frac{i}{b} \rfloor + 1} f(kb) c_{i-kb} = 0. \tag{4}$$

We recognize this problem as the linear Lagrange interpolation of $f(i)$ using values at $f(\lfloor i/b \rfloor b)$ and $f(\lfloor i/b \rfloor b + b)$. The next proposition gives a solution to these equations.

PROPOSITION 7.3. *Given integers n and b, such that b divides n, the equation* $f(i) - \sum_{k=0}^{n/b-1} f(kb) c_{i-kb} = 0$ *holds for all linear functions,* $f(x) = ax + b$, *if* $c_i = 1 - \frac{|i|}{b}$ *when* $i \in [-b, b)$.

PROOF. Setting $f(x) = 1$ and $f(x) = x$ in Eq. (4) yields two equations

$$1 = \sum_{k=\lfloor \frac{i}{b} \rfloor}^{\lfloor \frac{i}{b} \rfloor + 1} c_{i-kb}, \qquad i = \sum_{k=\lfloor \frac{i}{b} \rfloor}^{\lfloor \frac{i}{b} \rfloor + 1} kb c_{i-kb},$$

which are true when $c_i = 1 - \frac{|i|}{b}$. The general result ($f(x) = ax + b$) follows by linearity. □

We can verify that $\delta(kb) = 0$, using Proposition 7.2: when $l \notin \{0, -1\}$ then $lb \notin [-b, b)$, hence $c_{lb} = 0$. Otherwise, $c_{lb} = 1 - \frac{|lb|}{b}$, i.e., 1 when $l = 0$ and 0 when $l = -1$.

8. OVERLAPPED BIN BUFFERING FOR LOCAL MOMENTS: **OLA** BUFFERS

Overlapped Bin Buffering as described in the previous section can be used to buffer local moments as in subsection 7.2. In the special case of Overlapped Bin Buffering where the buffers are computed using Lagrange interpolation, we call the resulting data structure an OLA buffer.

Lagrange interpolation is a common technique discussed in standard Numerical Analysis references [Rao 2002, section 5.5]. In essence, given a function f and $M + M'$ samples of the function $f(m_1), f(m_2), \ldots, f(m_{M+M'})$, then

(1) we solve for the **unique** polynomial p of degree $M + M' - 1$ such that $p(m_1) = f(m_1), p(m_2) = f(m_2), \ldots, p(m_{M+M'}) = f(m_{M+M'})$;
(2) we evaluate the polynomial p at x and return this as the interpolated value.

It should be evident that if f is itself a polynomial of degree at most $M + M' - 1$, then the interpolation error will be 0; that is, $p(x) = f(x)$. We say that Lagrange interpolation of

order $M+M'-1$ reproduces polynomials of degree $M+M'-1$. Also, Lagrange interpolation is optimal, in the sense that it uses the smallest possible number of samples while reproducing polynomials of a given degree.

Given $M+M'$ samples $m_1,\ldots,m_{M+M'}$ of a function f, the *Lagrange formula* for the interpolated value at x is given by

$$f(x)=\sum_{k=m_1}^{m_{M+M'}}\left(\prod_{m=m_1;m\neq k}^{m_{M+M'}}\frac{x-m}{k-m}\right)f(k). \tag{5}$$

In Proposition 7.1, we introduced the function $\delta(i)$ given by

$$f(i)-\sum_{k=\lfloor\frac{i}{b}\rfloor-M'+1}^{\lfloor\frac{i}{b}\rfloor+M}c_{i-kb}f(kb)$$

which can also be written as

$$f(i)-\sum_{k=-M'+1}^{+M}c_{r-kb}f((k+\lfloor\frac{i}{b}\rfloor)b)$$

where $r=i-\lfloor\frac{i}{b}\rfloor b$. On the other hand, as a direct consequence of the Lagrange formula, choosing $m_1=b(\lfloor\frac{i}{b}\rfloor-M'+1),\ldots,m_{M+M'}=b(\lfloor\frac{i}{b}\rfloor+M)$ with $x=i$, we have

$$f(i)=\sum_{k=-M'+1}^{M}\left(\prod_{m=-M'+1,m\neq k}^{M}\frac{r-mb}{kb-mb}\right)f((k+\lfloor\frac{i}{b}\rfloor)b).$$

Now, by Lagrange's formula, if f is a polynomial of degree $M+M'-1$ and if

$$c_{r-kb}=\prod_{m=-M'+1,m\neq k}^{M}\frac{r-mb}{kb-mb},$$

then $\delta(i)=0$. When $M=1,M'=1$, Lagrange interpolation becomes equivalent to linear splines and is trivial to compute: if $k=0$, $c_r=\frac{b-r}{b}$ else if $k=1$, $c_{r-b}=\frac{r}{b}$; we conclude that $c_{r-kb}=1-\frac{|r-kb|}{b}$.

For $M+M'>2$, we can simply apply the formula

$$\begin{aligned}c_{r-kb}&=\prod_{m=-M'+1,m\neq k}^{M}\frac{r-mb}{kb-mb}\\&=\frac{\prod_{m=-M'+1,m\neq k}^{M}r/b-m}{\prod_{m=-M'+1,m\neq k}^{M}k-m}\\&=\frac{(-1)^{M-k}\prod_{m=-M'+1,m\neq k}^{M}r/b-m}{(M-k)!(k+M'-1)!}\end{aligned}$$

and precompute the coefficients once for b possible values of $r=0,\ldots b-1$ and $M+M'-1$ possible values of $k=-M'+1,\ldots,M$. This gives a total of $(M+M'-1)b$ coefficients.

The next lemma applies these results and says that Overlapped Bin Buffering efficiently buffers the first $M+M'$ moments when using coefficients derived from Lagrange interpolation. Hierarchical OLA with $b=\sqrt{n}$ is One-Scale OLA, thus we need not make an explicit distinction between the two.

LEMMA 8.1. *Overlapped Bin Buffering with overlap parameters $M, M' \geq 0$ allows $\delta(i) = 0$ for polynomials of degree $M + M' - 1$, if the coefficients c are chosen to be the Lagrange coefficients of degree $M + M' - 1$.*

8.1 Local Moments Using One-Scale OLA

We have already seen that each buffer value B_k in the OLA buffer is a sum over several bins, $B_k = \sum_{j=-Mb}^{M'b-1} c_j a_{j+kb}$ where the c_j are determined by Lagrange interpolation or directly as in subsection 7.2. By Lemma 8.1, if f is any polynomial of degree $M + M' - 1$, then local moment queries of the form $\sum_i f(i) a_i$ can be answered from the OLA buffer alone. However, consider function $f(x) = x$ over $[2, 10]$ and zero elsewhere. This function is used when computing the first-order query moment $\sum_{i=2}^{10} i a_i$. (See subsection 7.1.) Our approach must be refined to handle f and other useful functions.

An alternate viewpoint to the approach emphasizes how Lagrange interpolation provides an approximation h to function f. Knowing how h differs from f allows us to compensate for f's not being polynomial or (in section 11) allows us to bound the error from imperfect compensation. The details of this viewpoint follow.

By the proof of proposition 7.1, if we define

$$h(i) = \sum_{k=\lfloor \frac{i}{b} \rfloor - M' + 1}^{\lfloor \frac{i}{b} \rfloor + M} f(kb) c_{i-kb}$$

then $\sum h(i) a_i = \sum f(kb) B_k$. It is interesting to note that given some function f, we can compute h using a Lagrange polynomial *for each bin* as follows.

(1) Pick the bin, and suppose it comprises the cells in $[kb, (k+1)b)$.

(2) Obtain $M + M'$ data points by sampling f at $\{(k - M + 1)b, \ldots, kb, \ldots, (k + M' - 1)b, (k + M')b\}$.

(3) There is a unique polynomial $g_{(k)}$ of degree $M + M' - 1$ going through all those data points. Notice that $g_{(k)}$ is computed bin-wise.

Whenever f is polynomial of degree at most $M + M' - 1$ over $[(k - M + 1)b, (k + M')b]$, then $f = g$ over this same interval because the polynomial is unique. Moreover, the Lagrange coefficients satisfy Proposition 7.2 and thus $h(kb) = f(kb)$ for all k. Then, we can define a function by piecing together the bin-wise polynomials and because the Lagrange interpolant is unique, we have $h_{|[kb,(k+1)b)} = g_{(k)}$, since both sides of the equation are Lagrange interpolants of the same points. That is, there is a unique linear interpolation algorithm of degree $M + M' - 1$ over $M + M'$ data points, exact for polynomials of degree $M + M' - 1$. For our range queries, the query function f is pieced together from a polynomial (inside the range) and the polynomial $g(i) = 0$ (outside the range). Hence, $f(i) - h(i)$ will be zero except for two ranges of width $(M + M')b$, located around each of the range query's end

points. Moreover, $\sum h(i)a_i = \sum f(kb)B_k$ implies by Eq. (3) that

$$\begin{aligned}\sum_i \delta(i)a_i &= \sum_i f(i)a_i - \sum_{k=0}^{n/b} f(kb)B_k \\ &= \sum_i f(i)a_i - \sum h(i)a_i \\ &= \sum_i (f(i) - h(i))a_i.\end{aligned}$$

Therefore $\sum_i \delta(i)a_i$ can be computed in time $O(N^2b)$, accessing $M + M'$ bins from our external array.

Hence, Eq. (3) tells us how to answer the original query, $\sum_i f(i)a_i$, faster than the $\Omega(n)$ time required without precomputation. The query can be rewritten as the sum of $\sum_{k=0}^{n/b} f(kb)B_k$ and $\sum_i \delta(i)a_i$, (computed in time $O(n/b)$ and $O(N^2b)$, respectively). Therefore,we obtain a net reduction of the complexity from n to $n/b + N^2b$.

We note that the N^2b factor could be improved to Nb, if we precompute h for the b possible positions where an endpoint could fall within a bin, and for the N basic functions $1, i, i^2, \ldots, i^{N-1}$ we expect used with local moments. Each of these Nb values would have $\Theta(Nb)$ tabulated entries, leading to a storage complexity of $\Theta(N^2b^2)$, possibly too high for the small time savings except for tiny values of b. Yet small values of b would arise in the hierarchical setting that we shall next explore.

8.2 Local Moments Using Hierarchical OLA

We can further decrease the complexity by a hierarchical approach. We reduced the complexity from $O(Nn)$ to $O(Nn/b + N^2b)$. After the first transform, the cost is dominated by the computation of $\sum_{k=0}^{n/b} f(kb)B_k$. By the same method, we can reduce the complexity further to $n/b^2 + N^2 2b$. Because in-place storage is possible, this comes with no extra storage burden. Applying the method $\log_b n$ times reduces it down to $O(N^2 b \log_b n)$. The net result is similar to Bin Buffering, except that we are able to buffer the first two moments simultaneously, using a single buffer.

9. USING OVERLAPPED BIN BUFFERING FOR FAST LOCAL MOMENTS (OLA): ALGORITHMS AND EXPERIMENTAL RESULTS

This section puts the ideas of the previous section into practice, presenting more details of OLA and presenting an experimental analysis of OLA as an example of Hierarchical Bin Buffering. There are three fundamental operations on OLA buffers: building them initially, using them for fast queries, and finally, updating them when the underlying data changes. Each fundamental operation will, in turn, be described and its complexity analyzed. However, our complexity analysis is typical and ignores system-specific factors including the relative costs of various mathematical operators and the effects of the memory-access patterns on a computer's memory hierarchy. To show that the algorithm can be efficiently implemented, we we coded OLA in C++. The performance of our implementation completes the discussion of each fundamental operation.

To enable replication of our results, we next provide some details of our implementation and the test environment. Our experiments were conducted with N being even, and we chose $M = M' = \frac{N}{2}$; these constraints were imposed by our implementation. Our test platform was a Pentium 3 Xeon server with 2 GB RAM running the Linux operating system

(kernel 2.4), and the software was compiled using the GNU compiler (GCC 3.2 with -O2). For these experiments, we simulated arrays of any size by "virtual arrays" defined by $a_i = \sin(i)$: the function sin is chosen arbitrarily and the intent is that the access time to any one array element is a fixed cost (a calculation rather than a memory or disk access). Results are thus less dependent on current disk characteristics than would be otherwise possible. See section 9.4 for disk-based experiments.

For our experiments, array indices were always 64 bits, whereas stored values are 32-bit floating-point values. Unless otherwise specified, $n \approx 2^{30}$, giving us about 4 GB of virtual-array data; by "size", we refer to n, the number of array elements. As well, the buffer always fit within main memory and no paging was observed. We considered several different values of b: 32, 128, 1024, 2^{15} and 2^{20}. The first three values imply hierarchical OLA and can be justified by the ratio of main memory to external memory on current machines. The last two values imply one-scale OLA and fit the introduction's scenario that $\sqrt{n}$ would be an appropriate internal buffer size ($b = 2^{15}$), or fit a scenario where the user wants whatever gains can be obtained from a tiny buffer. We chose the value $b = 128$ as the "typical" value when one was needed.

For N, we experimented with values 2, 4, 8 and 16, with 4 deemed the typical value. Value 2 does not enable all the local moments that are likely to be used in practice, whereas we could not imagine any scenario where 16th or higher moments would be useful.

9.1 Computing the OLA Buffer

The construction of the OLA Buffer is possible using one pass over the external array, as illustrated in Algorithm 1.

Each buffer component is over a number of bins that depends linearly on the number of buffered moments. Similarly, as the size of the input array increases, we expect a linear increase in the construction time. The reason is that the number of buffered components increases linearly with n: although the number of buffer scales, β, increases with n, we have $\sum_{k=1}^{\beta} n/b^k \in \Theta(n)$. Each component has a computation cost that depends only on N, leading to an overall construction time of $O(Nn)$.

Of course, the storage required is inversely proportional to b: n/b. However, we do not expect the construction time to vary significantly with b as long as the buffer is internal (n/b is small). Indeed, when b grows, then the cost of computing each buffer element grows linearly and is proportional to Nb. On the other hand, the number of buffer components decreases with $1/b$. In total, the cost is roughly independent of b. If n/b grows substantially, then we might expect a slight time increase due to poorer memory performance on the large array. However, for all our experiments, the buffer size remained much smaller than system RAM. Experimental data to substantiate this is shown in Figures 4–5. The scale of Figure 4 magnifies what was less than a 7% difference between construction times, and we saw a small increase (5%) in construction time when b increased, which was not as anticipated. However, the point stands that buffer construction was not heavily affected by the choice of b.

Algorithm 1 OLA Buffer Computation

constants: bin size b, even number of buffered moments N and Lagrange coefficients c of degree $N-1$, β is the largest integer such that $n/b^{\beta} \geq N$.

function computeBuffer(a):
INPUT: an array a
OUTPUT: an array B (OLA buffer)
$B \leftarrow$ onestep$(a,0)$
for s = $0,1,\ldots,\beta-1$ **do**
  $B' \leftarrow$ onestep(B,s)
  for $k \in \{0,1,\ldots,size(B')-1\}$ **do**
    $B_{kb^{s+1}} \leftarrow B'_k$
  end for
end for

function onestep(a,s):
INPUT: an array a
INPUT: a scale parameter s
OUTPUT: an array B
Allocate $\left\lfloor \frac{size(a)}{b^{s+1}} + 1 \right\rfloor$ components in zero-filled array B
for $i = 0,1,\ldots,\lfloor size(a)/b^s \rfloor + 1$ **do**
  if i is a multiple of b **then**
    $B_{\lfloor i/b \rfloor} \leftarrow B_{\lfloor i/b \rfloor} + a_{ib^s}$
  else
    for $m = -\frac{N}{2}+1,\ldots,\frac{N}{2}$ **do**
      $B_{\lfloor i/b \rfloor + m} \leftarrow B_{\lfloor i/b \rfloor + m} + c_{-mb+i \bmod b} a_{ib^s}$
    end for
  end if
end for

9.2 Fast Local Moments Using the OLA Buffer

The algorithm for fast queries follows from Proposition 7.1: recall that we choose $M' = \frac{N}{2}, M = \frac{N}{2}$. As a first step, we have to compute $\sum_i \delta(i) a_i$ where

$$\delta(i) = f(i) - \sum_{k=\lfloor \frac{i}{b} \rfloor - \frac{N}{2}+1}^{\lfloor \frac{i}{b} \rfloor + \frac{N}{2}} f(kb) c_{i-kb}.$$

Then for scales $s = 1,\ldots,\beta$, we add $\sum_i \delta^{(s)}(i) B_{ib^{s-1}}$ where

$$\delta^{(s)}(i) = f(ib^s) - \sum_{k=\lfloor \frac{i}{b} \rfloor - \frac{N}{2}+1}^{\lfloor \frac{i}{b} \rfloor + \frac{N}{2}} f(kb^{s+1}) c_{i-kb}.$$

Finally, at scale $s = \beta$, we add to the previous computations $\sum_{k=0}^{n/b^{\beta-1}} f(kb^{\beta+1}) B_{kb^{\beta-1}}$.

The key to an efficient implementation is to know when $\delta(i)$ — or $\delta^{(s)}(i)$ — will be zero given a function f, so that we only sum over a number of terms proportional to Nb at each

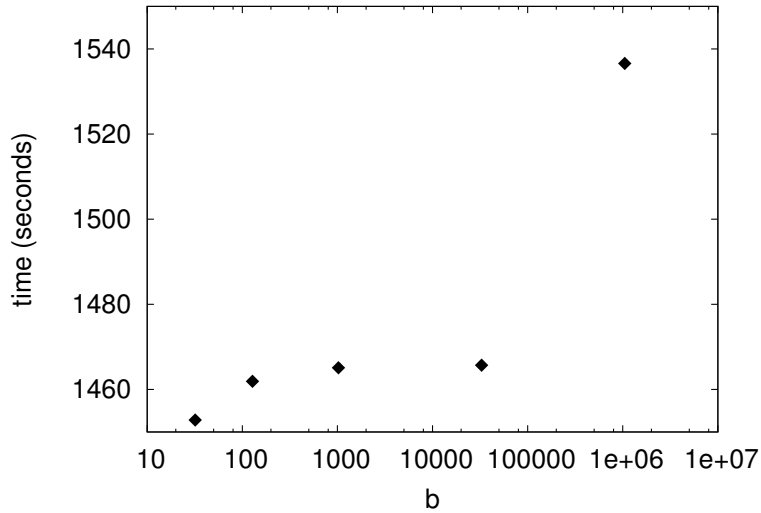

Fig. 4. Time in seconds for the construction of an OLA buffer, with $N = 4$ and varying values of b.

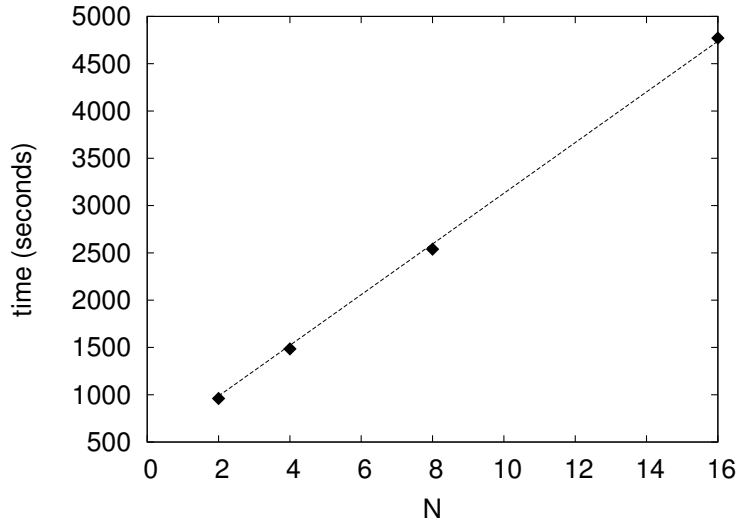

Fig. 5. Time in seconds for the construction of an OLA buffer, with $b = 2^{15}$ (one-scale OLA) and varying values of N. Times were almost identical for hierarchical OLA using $b = 128$. Curve $t(N) = 268N + 450$ (shown) fits the points well.

scale. In other words, we need to do a lazy evaluation. Suppose that f is a polynomial of degree at most $N-1$ for N even over the interval $[p,q]$ and the overlap parameters are $M = \frac{N}{2}, M' = \frac{N}{2}$, then $\delta^{(s)}$ may be nonzero only over intervals $[p',p'']$ and $[q',q'']$, where $p' = (\lfloor \frac{p}{b^{s+1}} \rfloor - \frac{N}{2})b$, $p'' = (\lfloor \frac{p}{b^{s+1}} \rfloor + \frac{N}{2})b$, $q' = (\lfloor \frac{q}{b^{s+1}} \rfloor - \frac{N}{2})b$ and $q'' = (\lfloor \frac{q}{b^{s+1}} \rfloor + \frac{N}{2})b$. The

complete pseudocode is given in Algorithm 2.

Algorithm 2 Moment Computation using the OLA Buffer

constants: bin size b, even number of buffered moments N and Lagrange coefficients c of degree $N-1$, β is the largest integer such that $n/b^{\beta} \geq N$.

function query(a, B, f, p,q):
INPUT: an array a
INPUT: an OLA buffer B
INPUT: a function f which is a polynomial of degree at most $N-1$ over $[p,q]$ and zero elsewhere
OUTPUT: returns $S = \sum_{i=0}^{n-1} f(i)a_i$
$S \leftarrow 0$
for $i \in$ candidates$(size(a), 0, p, q)$ **do**
  $S \leftarrow S + (f(i) - \sum_{k=\lfloor i/b \rfloor - \frac{N}{2}+1}^{\lfloor i/b \rfloor + \frac{N}{2}} f(kb)c_{i-kb})a_i$
end for
for $s = 1, \ldots, \beta$ **do**
  for $i \in$ candidates$(size(B)/b^{s-1}, s, p, q)$ **do**
    $S \leftarrow S + \left(f(ib^s) - \sum_{k=\lfloor \frac{i}{b} \rfloor - \frac{N}{2}+1}^{\lfloor \frac{i}{b} \rfloor + \frac{N}{2}} f(kb^{s+1})c_{i-kb} \right) B_{ib^{s-1}}$
  end for
end for
$S \leftarrow S + \sum_{k=0}^{size(B)/b^{\beta-1}} f(kb^{\beta+1})B_{kb^{\beta-1}}$

function candidates($size$,s,p,q):
OUTPUT: the union of

$$\left\{ \max\left(\left(\frac{p}{b^{s+1}} - \frac{N}{2} \right) b, 0 \right), \ldots, \left(\frac{p}{b^{s+1}} + \frac{N}{2} \right) b - 1 \right\}$$

and

$$\left\{ \left(\frac{q}{b^{s+1}} - \frac{N}{2} \right) b, \ldots, \min\left(\left(\frac{q}{b^{s+1}} + \frac{N}{2} \right) b - 1, size \right) \right\}.$$

Queries are $O(b\beta)$ where $\beta = \log_b n$, so when b increases the algorithm's running time increases in proportion to $\frac{b}{\log b}$. Therefore, because the buffer size is given by n/b, the algorithm becomes slower as the size of the buffer is reduced. Experimentally, this was measured by randomly selecting, with replacement, 2000 of the $\binom{n}{2}$ different non-empty ranges. More precisely, we choose two uniformly distributed random numbers a and b and pick the interval $[\min(a,b), \max(a,b))$. We set N to a "typical value" of 4, and timed the 2000 sums and 2000 first moments for various b values[2].

Results are shown in Figure 6; we also plotted the function $t(b) = b/(5000 \ln b)$. The measured running time appears to grow no faster than $t(b)$.

[2]When $N^2 b$ was large, we tested only $\lfloor \frac{800{,}000}{N^2 b} \rfloor$ cases, to keep test times reasonable.

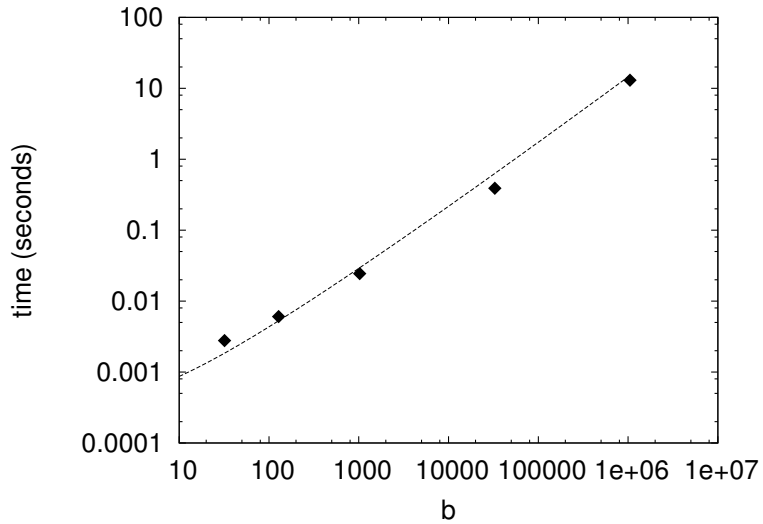

Fig. 6. Average time (in seconds) for the computation of many randomly selected range sums for $N = 4$ and various values of b. Function $t(b) = b/(5000 \ln b)$ is also shown.

Note that the time for a range query is affected somewhat by the length of the range l, in that the number of buffer elements B_i accessed will be approximately l/b for one-scale OLA. (For hierarchical OLA the relationship between the number of buffers accessed and the range length is much more complex.) As well, for hierarchical OLA, the number of hierarchical levels processed will also depend on the precise positioning of the range's endpoints. To see these effects, we plotted the time[3] versus the range size. Results for One-Scale OLA (see Figure 7) are as expected: the time was dominated by the (unvarying) work done around the range's endpoints. There was a small additional contribution coming from the number of buffer values accessed, which showed up as a slight upward slope on the cluster. First moments and sums behaved similarly.

[3]Timed on a slightly faster Pentium 4 machine with 512 MB RAM, running a Linux 2.4 kernel that had been patched to supply high-resolution timings and hardware performance counts via PAPI[Browne et al. 2000].

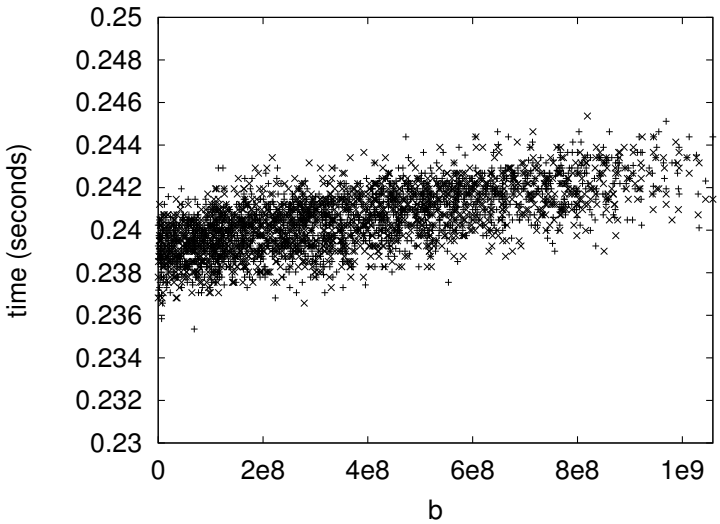

Fig. 7. Average time (in seconds) versus range length for the computation of 1,525 randomly selected first moments (x) and 1,525 sums (+) for $N = 4$ and $b = 2^{15}$. (One-Scale OLA.)

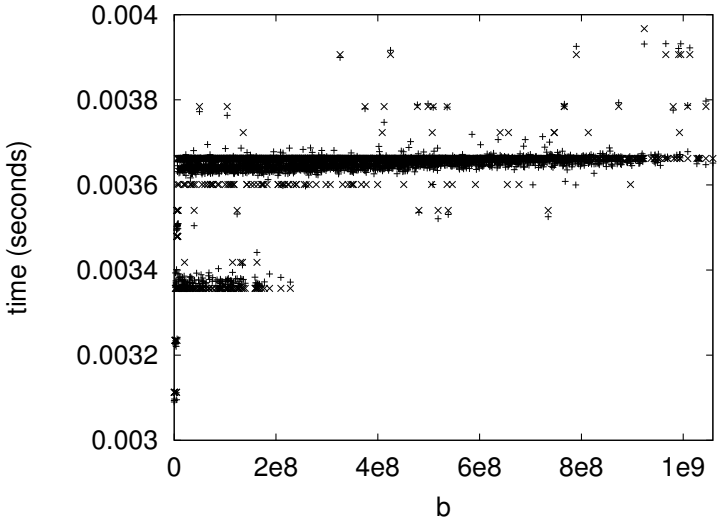

Fig. 8. Time in seconds versus range length for 2,000 randomly selected first moments (x) and 2,000 sums (+) with $N = 4$ and $b = 128$ (4-Scale OLA).

The situation is more complex for hierarchical OLA (see Figure 8), where the positioning of the range determines the number of buffer values and where the positioning of each endpoint determines how many external array elements are accessed and determines how

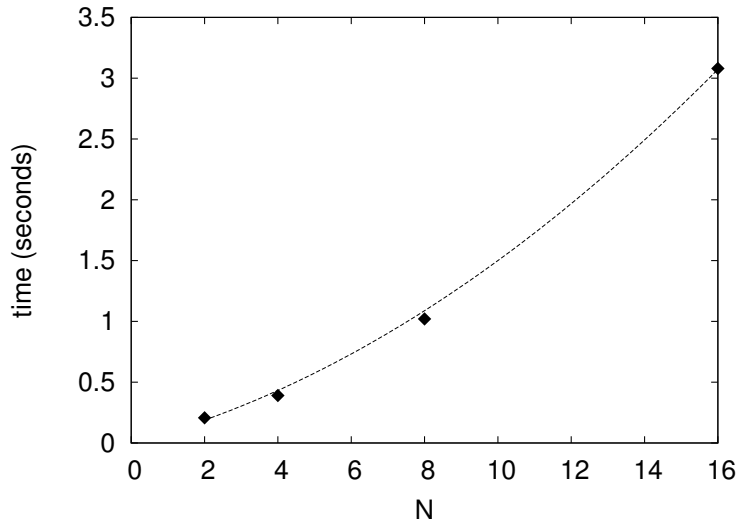

Fig. 9. Average time per query (seconds) versus N for randomly selected range sums with $b = 2^{15}$ (one-scale OLA). (The points fit $t(N) = .007N^2 + .08N$ well.)

many hierarchical scales need to be considered for the region surrounding each endpoint. Since the running times are smaller, it is perhaps not surprising that the data appears noisier.

We can show that query times for hierarchical OLA grow quadratically as the number of moments buffered is increased. For one-scale OLA, queries have two main sources of cost: first, the cost from computing $\sum_k f(k)B_k$, where f is piecewise 0 or an $N-1^{\text{st}}$ degree polynomial, which we evaluate at a cost of $\Theta(N)$ per point within the range of the query. The expected range of our queries is long, so this cost is significant. The second cost of our queries comes from a $\Theta(N^2)$ calculation done around the range's endpoints. Therefore, for our small values of N, the total cost includes both a large N^2 as well as a large linear component. From a theoretical point of view, however, the growth is $\Theta(N^2)$ and is dominated by the endpoint computations. (See Figures 9 and 10). Hence, it might be detrimental to buffer many more moments than we require. However, the number of moments has no effect on the space complexity, unlike the bin size, b. Therefore, for large enough arrays, even if we buffer many moments, the OLA approach will still be several order of magnitude faster than unbuffered queries.

The OLA approach was not sensitive to the query: range sums or first moments were measured to take almost exactly (within 1%) the same time. Therefore, these results are not plotted.

Based on the theoretical analysis, OLA can permit huge query speedups, given extreme values for parameters such as the relative speeds of internal versus external memory, amount of memory allocated to the buffer, and so forth. However, we need good speedups for "reasonable" parameters. From our experiments, it is evident that buffered arrays were considerably faster, and random queries that averaged about 71.6 s without buffering could be answered in 0.390 s or 0.00605 s when the 4 GB dataset was buffered with 128 kB or 32 MB (using N=4). This corresponds to respective speedups of 184 and 11800. The

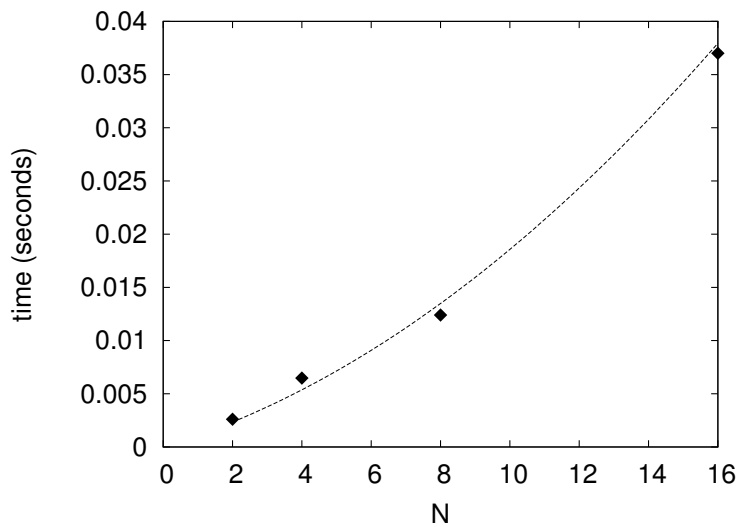

Fig. 10. Average time per query versus N for 2000 randomly selected range sums with $b = 32$ (3- or 4-Scale OLA). The running time fits $t(N) = (N^2 + 11.7N)/11700$ well.

construction time of approximately 1500 s means a total construction+query break-even is achieved after about 21 queries.

9.3 Updating the OLA Buffer

To update the buffer, we can consider how the buffer was originally constructed: it was computed from the data source and then, in a hierarchical manner, buffers were computed from the previous buffer and stored in place. Recall, for instance, that in the OLA buffer, values of the second-scale buffer are stored in indices $b, 2b, \ldots, (b-1)b, (b+1)b, (b+2)b, \ldots, (2b-1)b, (2b+1)b, \ldots$ whereas the values of the third-scale buffer are stored at $b^2, 2b^2, \ldots, (b-1)b^2, (b+1)b^2, \ldots$ and so forth. We define the cells at scale s as those having an index divisible by b^{s-1}, with the expository convention that "scale 0" refers to entries in the external array. Our updates propagate changes from smaller scales to larger scales. (See Algorithm 3.)

To understand the update algorithm acting upon this hierarchical buffer with in-place storage, it is helpful to consider the "is computed from" relation between cells, which forms a directed acyclic graph (dag). Showing each cell at every scale to which it belongs, coloring (black) the largest scale for each cell, and focusing only on the part of the dag that needs to be updated, we obtain Figure 11. The black vertices at scale s (for $s \geq 1$) in the dag correspond to indices i such that $\lfloor \frac{i}{b^{s-1}} \rfloor$ is not divisible by b, that is, cells that do not belong to scale $s+1$. The uncolored vertices belong to scale $s+1$ and in-place storage means that an update affects the black node beneath it (except for the largest scale since the algorithm terminates). The portion of the dag that is reachable from the changed cell (at scale 0) is called the *update dag*.

From this, we see that the update cost is linear with the height of the update dag. To prove it, we first observe that we can bound, independently of the height of the dag (given

by $\beta \sim \log_b n$), the number of cells per scale that need to be updated.

PROPOSITION 9.1. *By Algorithm 3, given an update of one cell in the external array, updates are propagated from one scale to another over at most* $2N$ *cells. In other words, the update dag (as in Figure 11) has at most* $2N$ *nodes at each level.*

PROOF. Let $l_{(s)}$ be the difference in indices between the last modified buffer cell and the first modified one (ordering is by indices) at the end of step s in Algorithm 3. In other words, $l_{(s)}$ is the "range" of the modified cells at step s. By convention, $l_{(0)} = 0$. From the algorithm, we see that $l_{(s)} \leq l_{(s-1)} + Nb^s$. Hence, we have that $l_{(s)} \leq 2Nb^s$ so that $l_{(s)}/b^s = 2N$. Hence, each time we move from one scale to another, at most $2N$ cell values are modified. □

This proposition tells us that the middle `for` loop in Algorithm 3 has at most $2N$ steps; since we do $O(N)$ operations within each, the update complexity is $O(N^2\beta)$.

Correctness of the algorithm is straightforward and relies on the fact that index i is processed only at its largest scale (see the first `if` statement). At this time, all updates to i will have been completed.

The effect of N on computational cost as measured experimentally is given by Figure 13. The observed relationship appears linear, but the three collinear points are misleading; for $N = 2$ and $N = 4$, we had $\beta = 4$. However, for $N = 8$ and $N = 16$, we had $\beta = 3$. As well, since only $O(N)$ hashmap entries are created (and then updated $O(N)$ times each), if hashmap-entry creation is expensive, then the running times will contain a large linear component in N.

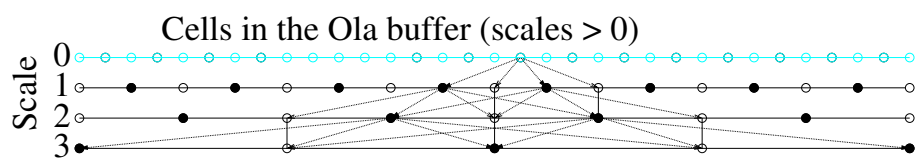

Fig. 11. Update dag for OLA buffer with $M = M' = 2$ ($N = M + M' = 4$) and $b = 2$ (see Algorithm 3). Each column with an entry at Scale 1 corresponds to a buffer cell, whereas entries at Scale 0 are in the external array.

If the original array was not dense, that is, if most components were zero, then it can be more efficient to construct the buffer starting with a zero buffer and then adding each non-zero value as an update. Because the cost of each update is $O(N^2\beta)$, if there are $d(n)$ non-zero values in the original array, then the complexity of building the buffer through updates is $O(d(n)N^2\beta)$. This is asymptotically better than Algorithm 1 whenever $d(n) \in o(n/N\beta)$. Experimentally, for $N = 4$ and $b = 128$, we made 200k random updates in about 12 seconds. Thus, even if data items in a sparse set were added in an unordered manner, it would be faster to build the buffer through updates if it had about 24 million or fewer elements, or a density of $\frac{2.4\times10^7}{1\times10^9}$ = 2.4% or less. Since update time decreases rapidly with b whereas the time for Algorithm 1 is almost independent of b, once $b \geq 2^{15}$ incremental construction is a reasonable alternative to Algorithm 1 for any data set.

Algorithm 3 Updating the OLA Buffer.

constants: bin size b, even number of buffered moments N and Lagrange coefficients c of degree $N-1$, β is the largest integer such that $n/b^{\beta} \geq N$.

function update (B, j, Δ):
INPUT: an index j in the original array a
INPUT: an OLA buffer B over the array a
INPUT: the change Δ in the value of a_j
RETURN: modifies B
$deltas$ is a (hash) map {assume 0 for unassigned values}
$deltas_j \leftarrow \Delta$
for $s = 0, \ldots, \beta$ **do**
  Let keys($deltas$) be the set of keys for the hash table $deltas$ at this point
  { Invariant: keys($delta$) contains only indices at scale s}
  for $i \in$ keys($deltas$) **do**
    if $\lfloor \frac{i}{b^s} \rfloor$ is not divisible by b **then**
      {Process i because s is its largest scale}
      $\delta \leftarrow deltas_i$
      for $m = -\frac{N}{2}+1, \ldots, \frac{N}{2}$ **do**
        $deltas_{(\lfloor \frac{i}{b^{s+1}} \rfloor + m)b^{s+1}} \leftarrow deltas_{(\lfloor \frac{i}{b^{s+1}} \rfloor + m)b^{s+1}} + c_{-bm+(\lfloor \frac{i}{b^s} \rfloor \bmod b)}\delta$
      end for
      if i is divisible by b **then**
        { Only (possibly) j is not a multiple of b}
        $B_{i/b} \leftarrow B_{i/b} + deltas_i$
      end if
      remove key i from $deltas$
    end if
  end for
end for
{ Cells belonging to scales $\beta+1$ and above are still in $delta$ and need to be added to the buffer (see uncolored nodes at the last level of Figure 11).}
for $i \in$ keys($deltas$) **do**
  $B_{i/b} \leftarrow B_{i/b} + deltas_i$
end for

A key point is that updates to the buffer get progressively less expensive as b goes up and the size of the buffer goes down. Figures 12 and 13, as well as Table II provide experimental evidence of these claims.

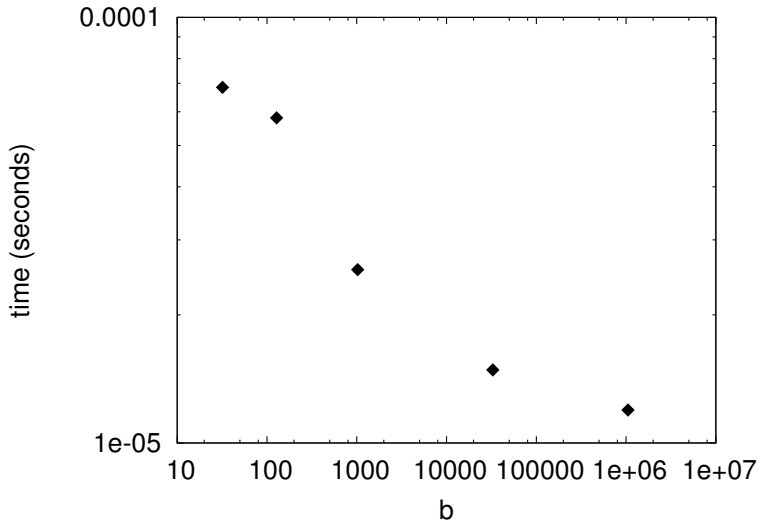

Fig. 12. Average time versus b for 200,000 random updates with $N = 4$

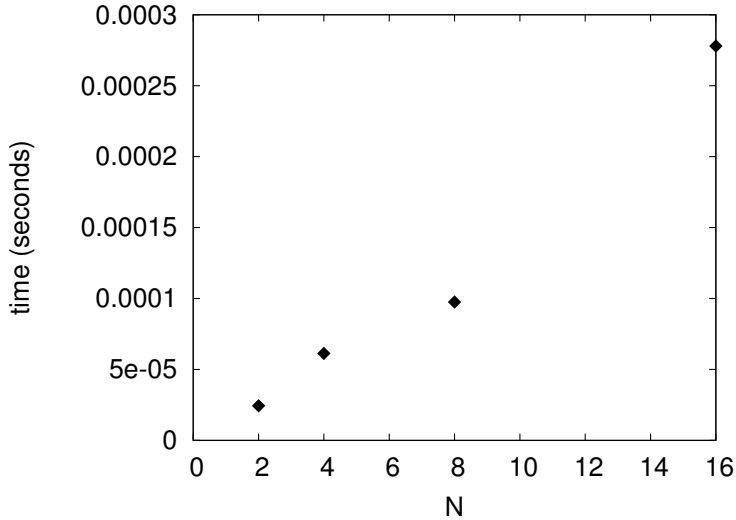

Fig. 13. Average time versus N for 200,000 random updates with $b = 128$ (3- or 4-Scale OLA).

9.4 External Memory

The use of "virtual arrays" in the previous section allowed us to abstract away from the specific details of current memory-system hardware. However, one might abuse such simplified models, thus incorrectly predicting good practical performance for algorithms that

Table II. Linear relationship observed between β and update time. $N = 4$ and average time was over 200,000 random updates.

b	β	time (μs)	time/β
32	5	68.5	13.7
128	4	58.1	14.5
1024	2	25.6	12.8
2^{15}	1	14.9	14.9
2^{20}	1	12.0	12.0

make irregular and non-local accesses to disk. Nevertheless, our algorithms for buffer construction and queries tend to have good locality. For instance, with queries in One-Scale OLA, two consecutive groups of indices (around either endpoint of the queried range) are accessed. Experiments to support our claims were derived using memory-mapped files. Unfortunately, due to our experimental setup (mainly a 32-bit address space), we were forced to choose a smaller value of $n \approx 2^{28}$ elements, or about 1 GB of data. These experiments were performed on a computer with 512 MB of RAM, and since much I/O was anticipated (and observed), we took wall-clock times while the system ran in single-user mode.

Repeating the experiments in which we timed the construction of an OLA buffer with $N = 4$ and varying b, we obtained the results shown in Figure 14. We note that the discrepancy for $b = 128$ does not seem to be an error: it was repeatable. Except for this one value of b, we see that changes in b affected construction time by less than 10%. The large discrepancy at $b = 128$ apparently came from the operating system and system libraries[4].

By conducting the experiments of subsections 9.1 to 9.3 with virtual arrays, we avoided many secondary system-level effects that might have obscured our results. But it is useful to compare the timings obtained with (realistic) memory-mapped array versus those from our synthetic virtual arrays. For reference, with virtual arrays and $n \approx 2^{28}$, a construction time of about 250 s was obtained with $N = 4$, for all values of b. We see that the virtual array lead to a construction time that was approximately three times longer.

We also timed random range-sum queries using our memory-mapped array (see Figure 15). For comparison, similar random queries were also computed directly from the external array, without using the OLA buffer at all. Despite the good locality of the obvious algorithm for this task, with $N = 4$ and $b = 128$, OLA answered the query less than .014 seconds, versus 6.3 seconds when no buffer was used: a speedup of over 400. For reference, a virtual array lead to query times that were approximately 11% *slower* than with the memory-mapped implementation for $b = 2^{15}$, $N = 4$ but about 75% faster for $b = 128$, $N = 4$.

Comparing Figure 15 to Figure 6, we observe that with the memory-mapped array, the

[4]Using the same hardware, but with the Linux kernel upgraded to version 2.6.20, glibc to version 2.5, and the GNU C++ compiler to version 4.1.2, we obtained different results: the cases $b = 32$, $b = 128$ and $b = 2^{20}$ were similar. (The median time of 25 runs for $b = 128$ was no more than 5% larger than the medians of the other two cases.) The cases of $b = 1024$ and $b = 32768$ were similar, their medians being slightly less than 20% faster than $b = 128$. Repeating tests for $N = 16$, all cases except $b = 2^{20}$ were similar to one another, whereas $b = 2^{20}$ was approximately 50% slower than the others. The effects of varying N and b are described in Section 9.1; we conjecture that some combinations of N and b produce page-access streams that are easier for the operating system to handle efficiently. Further investigation is outside the scope of this paper.

query time is not as sensitive to differences in b, when b is small. Presumably this is due to blocking on disks: even when b is small, at least an entire virtual memory page or disk block needs to be dedicated to the area around each endpoint of the query range.

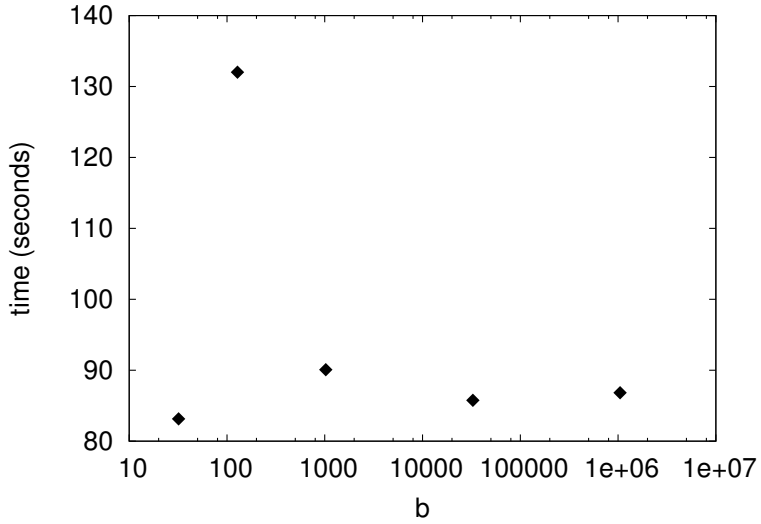

Fig. 14. Time to construct an OLA buffer ($N = 4$) from a memory-mapped disk file, versus b.

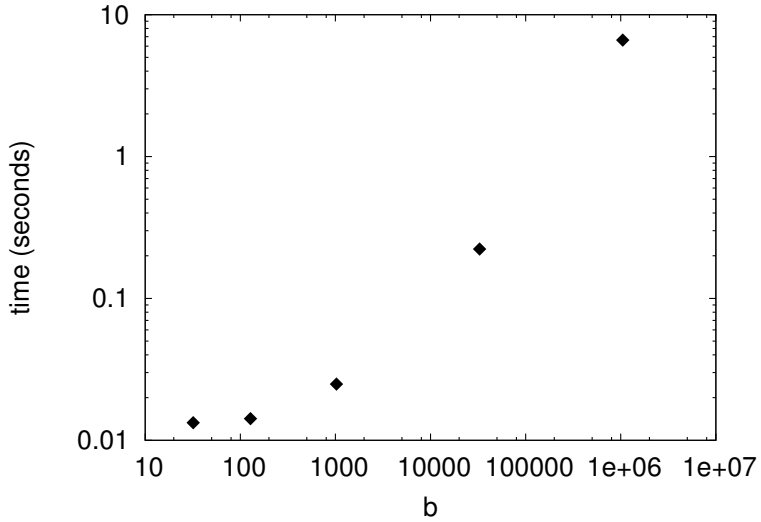

Fig. 15. Average time to answer a random range-sum query from an OLA buffer ($N = 4$) versus b. A memory-mapped file with $n \approx 2^{28}$ 4-byte floating-point numbers was used.

10. OLA VERSUS BIN BUFFERING

Assume that we are given the task of buffering N moments with a fixed amount of internal memory K over a very large array of size n. Recall that given an array of size n and a buffer size b, OLA will use a buffer of size $n/b+1$. Hence, OLA would lead to bins of size $b = n/(K-1) \approx n/K$ whereas BIN BUFFERING would use larger bins of size $b' = Nn/K$.

Assume we can read bins of size b from external memory with a fixed cost of E_b units of time, and we can access internal memory cells with a cost of 1 unit of time. For simplicity, we also assume that $Nb < E_b$, which seems likely given the small values of N anticipated. One-scale OLA and BIN BUFFERING have then exactly the same complexity, that is, queries have worst-case complexity $O(NE_b + K)$. The hierarchical versions also have similar complexity to one another.

However, not all queries have the same cost: the two algorithms are **not** equivalent. BIN BUFFERING will support $\binom{K/N}{2}$ ranges without *any* access to the external array: all range queries from bin edges to bin edges can be answered entirely from the buffer. For instance, $\sum_{i=2K/N}^{10K/N-1} a_i$ can be answered by summing 8 buffer elements.

However, for some applications, we might be interested in how well we can approximate the query without access to the external array. This is especially important in applications such as visualization, where a very fast initial approximation is valuable.

To explain why OLA is more competitive in providing good approximations using only the buffer, take the case where $N = 2$ and assume that the values in the external array are uniformly bounded in absolute value by κ. That is, $|a_j| \leq \kappa$ for all j. Recall that we assume that the internal buffer has a size of K. Then consider range sums such as $\sum_{j=k}^{l} a_j$. The largest error made by BIN BUFFERING is bounded by $2b'\kappa = 4\kappa n/K$ since we miss at most one bin at each end (whose total value is at most $b'\kappa$). We can reduce this bound to $2\kappa n/K$ by choosing to add bins whenever they are more than half occupied. On the other hand, the largest error that OLA can make is bounded by $2\kappa(b/2) = \kappa n/K$. Indeed, the worst error is reached when range edges match with bin edges. In that case, we wrongly take a full bin at each end. Actually, due to the linear decrease in c_i values, the more distant values in these bins are weighted lightly; this leads to the $b/2\kappa$ bound on each bin. Hence, OLA is twice as accurate for estimating range sums from the buffer when $N = 2$. Irrespective of N, the error for OLA is more than bounded by $2b\kappa = 2\kappa n/K$. However, the error made by BIN BUFFERING can be as bad as $b'\kappa = N\kappa n/K$, even taking into account the possible improvement one gets by including bins that are more than half used. In other words, as N grows, the worst-case error made by BIN BUFFERING grows linearly, unlike OLA. This result applies to hierarchical versions of these algorithms as well.

Hence, one might want to look at the case $N = 8$. This is not unreasonable in a visualization setting where the user can set the degree of the polynomials to be fitted. In such a case, BIN BUFFERING has very large bins (8 times larger than OLA) which might be undesirable: the approximation power of BIN BUFFERING for range sums is at least 4 times lower than OLA because of the much larger bins.

11. APPROXIMATE QUERIES USING OLA

We have seen that OLA can have a competitive advantage when we are interested in getting approximate queries out of the memory buffer. Indeed, OLA supports a wide range of query types using a single memory buffer and relatively small bins. However, as with wavelet-based techniques [Vitter et al. 1998; Chakrabarti et al. 2001; Schmidt and Shahabi

2002], OLA can support incrementally better estimates. With wavelet-based methods, one gets approximations by selecting the most significant wavelet coefficients. However, this approach calls for storing all coefficients in order to quickly answer queries within a user-specified error bound. This would be unacceptable for many applications: the wavelet buffer is as large as the external array itself. Incrementally better OLA approximations can be computed by first using the internal buffer, and then adding bins one by one. Indeed, for a given range query, the OLA algorithms involve many bins at both ends of the range. However, only the first few bins have a significant contribution.

Recall subsection 8.1, in which we showed that $\sum f(kb)B_k$ was given by $\sum h(i)a_i$ where h is a Lagrange interpolation of the range query function f. Only when the function f goes from a polynomial to 0 is there a difference between the target range function f and the range function h estimated by bin-wise Lagrange interpolation. However, the error made by Lagrange interpolation also diminishes as we move away from the bin containing the edge of the range. In many cases, it might be sufficient to take into account only 1 or 3 bins near the edge (at each endpoint of the range). For example, consider $N = 4$, $b = 1024$ with $f(x) = 1$ for $x > x_0$ and 0 otherwise. A numerical evaluation shows that using only one bin at each endpoint instead of the required 3 will take care of 86% of the error when range edges are in the middle of a bin. The result is more significant for larger N, for example, for $N = 16$, we need 15 bins at each endpoint for a complete evaluation; however, if we use only 5 centered, we take care of 97% of the error. (See Figure 16.) In short, OLA can provide wavelet-like progressive evaluation of the queries simply by querying fewer bins in the external array.

We can analyze more mathematically the relationship between the number of bins used at each end of the range and the error. First note that bin-wise Lagrange interpolation is linear: if we interpolate the sequence $\{0,0,0,1,0\}$ and the sequence $\{0,0,0,0,2\}$, then the sum of the two interpolants is just the interpolation of the sequence $\{0,0,0,1,2\}$. Hence, it is sufficient to consider only one non-zero sample value at any given time. We proceed to show that the contribution of a sample value $f(kb)$ to bin-wise Lagrange interpolation decays quickly past one bin. Let $N = M' + M$ be fixed, and consider the interpolation of the sequence $x_0 = 1$, $x_{bi} = 0$ for all $i \neq 0$. We can then consider the interpolant h in the kth bin defined by the interval $[bk, bk+b)$. The following polynomial (refer to Eq. (5)) describes h:

$$\frac{\prod_{i=M-1,i\neq k}^{M'}(x-kb+bi)}{\prod_{i=M-1,i\neq k}^{M'}(-kb+bi)}$$

where $x \in [bk, bk+b)$. The formula is only valid for $-M' \leq k \leq M-1$; elsewhere h is identically zero. Clearly the denominator will increase sharply in absolute value as k grows. We show that the numerator is non increasing in k. Setting $y = x - kb \in [0,b]$, we have $\prod_{i=M-1,i\neq k}^{M'}(x-kb+bi) = \prod_{i=M-1,i\neq k}^{M'}(y+ib)$. However, $\prod_{i=M-1,i\neq k}^{M'}(y+ib) = \frac{\prod_{i=M-1}^{M'}(y+ib)}{(y+kb)}$ and because $y \in [0,b]$, $y+kb \in [kb, kb+b]$ and so, the numerator goes down in amplitude as $1/k$. On the other hand, the denominator in absolute value, $(b(-k+M-1))\cdots(b)(b)\cdots(b(k+M')) = b^{M+M'}(M-1-k)!(k+M')!$. Setting $\lambda = k+M'$, we have $b^{M+M'}(M-1-k)!(k+M')! = b^{M+M'}(M+M'-1-\lambda)!\lambda! = b^{M+M'}(N-1-\lambda)!\lambda!$ which has a rate of increase of starting at $(N+2)/(N-2)$ and rising with k. Hence, the amplitude of the polynomial decreases faster than exponentially as k increases.

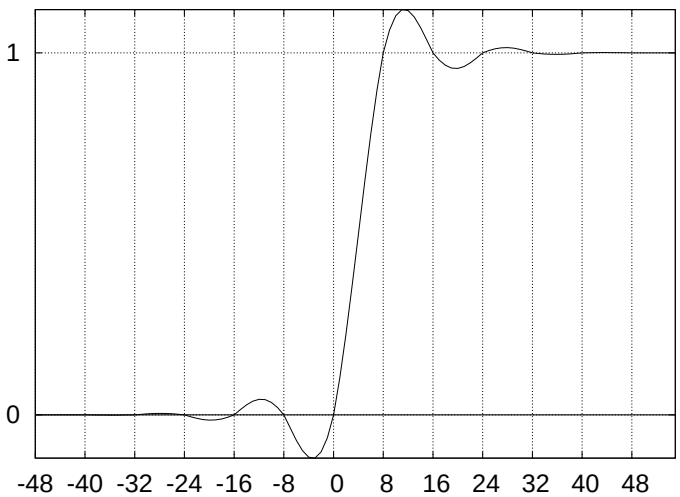

Fig. 16. Given a step function f going from 0 to 1, we show that the Lagrange interpolation h is quite close to f as we move away from the discontinuity. In this figure, $N = 16$ and $b = 8$.

It is difficult to compare the progressive approximation we get using this approach with related wavelet-based ones. Wavelet-based algorithms do not use the original array as a data source when answering queries and thus, they have much larger storage requirements. For large-scale applications, approximate queries are required for wavelet-based algorithms because storing all the coefficients is unthinkable whereas OLA has progressive approximate queries as an added option.

12. CONCLUSION AND FUTURE WORK

This paper has considered bin-buffering algorithms and showed that using a hierarchical approach, highly scalable algebraic queries were possible even with a small buffer. Using overlapped bins, we have shown that we could buffer several local moments simultaneously and use much less storage than wavelet-based approaches while still supporting progressive queries and very scalable queries and updates.

In short, we showed that N local moments could be buffered using only a single real-valued buffer: using bins of a fixed size irrespective of N. Other types of range queries could also be grouped and buffered efficiently together [Deligiannakis and Roussopoulos 2003]. By a direct product [Lemire 2002], Hierarchical Bin Buffering and therefore the OLA approach can be generalized to the multidimensional case.

Some implementation issues were not addressed. For example, many forms of buffering using finite-accuracy floating-point numbers are susceptible to significant numerical errors.

Finally, the source code used for the production of this paper is freely available [Lemire and Kaser 2007].

REFERENCES

ALON, N., MATIAS, Y., AND SZEGEDY, M. 1996. The space complexity of approximating the frequency moments. In *STOC'96*. ACM Press New York, NY, USA, 20–29.

BROWNE, S., DONGARRA, J., GARNER, N., HO, G., AND MUCCI, P. 2000. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications 14,* 3, 189–204.

CHAKRABARTI, K., GAROFALAKIS, M., RASTOGI, R., AND SHIM, K. 2001. Approximate query processing using wavelets. *The VLDB Journal 10,* 2-3, 199–223.

CLEVELAND, W. AND LOADER, C. 1995. Smoothing by local regression: Principles and methods. Tech. rep., AT&T Bell Laboratories.

CODD, E. F., CODD, S., AND SALLEY, C. 1993. Providing OLAP (On-line Analytical Processing) to user-analysts: An IT mandate. Tech. rep., E. F. Codd & Associates.

DELIGIANNAKIS, A. AND ROUSSOPOULOS, N. 2003. Extended wavelets for multiple measures. In *SIGMOD*. ACM Press, 229–240.

GEFFNER, S., AGRAWAL, D., ABBADI, A. E., AND SMITH, T. R. 1999. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *ICDE'99*. 328–335.

GRAY, J., BOSWORTH, A., LAYMAN, A., AND PIRAHESH, H. 1996. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and subtotals. In *Proc, 1996 ICDE*. 131–139.

HO, C.-T., AGRAWAL, R., MEGIDDO, N., AND SRIKANT, R. 1996. Range queries in OLAP data cubes. In *ACM SIGMOD*. 73–88.

IEC. 1999. Letter symbols to be used in electrical technology — part 2: Telecommunications and electronics. Tech. Rep. IEC 60027-2 Second Edition, International Electrotechnical Commission.

JAHANGIRI, M., SACHARIDIS, D., AND SHAHABI, C. 2005. SHIFT-SPLIT: I/O efficient maintenance of wavelet-transformed multidimensional data. In *SIGMOD '05*. 275–286.

LEMIRE, D. 2002. Wavelet-based relative prefix sum methods for range sum queries in data cubes. In *CASCON'02*.

LEMIRE, D. 2007. A better alternative to piecewise linear time series segmentation. In *SDM'07*.

LEMIRE, D. AND KASER, O. 2007. Hierarchical bin buffering library in C++. `http://code.google.com/p/hierarchicalbinbuffering/`, last checked on 15/7/2007.

LI, B.-C. AND SHEN, J. 1992. Fast calculation of local moments and application to range image segmentation. In *Int. Conf. Pattern Recognition*. 298–301.

MOERKOTTE, G. 1998. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB'98*. 476–487.

PATTERSON, D. 2003. A conversation with Jim Gray. *ACM Queue 1,* 4 (June), 6–7.

POON, C. 2003. Dynamic orthogonal range queries in OLAP. *Theoretical Computer Science 296,* 3, 487–510.

RAO, S. S. 2002. *Applied Numerical Methods for Engineers and Scientists*. Prentice Hall.

SCHMIDT, R. R. AND SHAHABI, C. 2002. Propolyne: A fast wavelet-based algorithm for progressive evaluation of polynomial range-sum queries. In *Conference on Extending Database Technology*. 664–681.

SCOTT, D. AND SAGAE, M. 1997. Adaptive density estimation with massive data sets. In *ASA, Statistical Computing Section*. 104–108.

SILVA, C., CHIANG, Y., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization'02 Course Notes*.

VITTER, J. S. 2002. *Handbook of massive data sets*. Kluwer Academic Publishers, Chapter External memory algorithms, 359–416.

VITTER, J. S., WANG, M., AND IYER, B. 1998. Data cube approximation and histograms via wavelets. In *CIKM*. ACM Press, 96–104.

ZHOU, F. AND KORNERUP, P. 1995. Computing moments by prefix sums. Tech. Rep. PP-1995-31, University of South Denmark.