Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Διαχείριση Ρευμάτων Δεδομένων

Διδακτορική Διατριβή

του

Δημήτρη Σαχαρίδη

Διπλωματούχου Ηλεκτρολόγου Μηχανικού &
Μηχανικού Υπολογιστών Ε.Μ.Π. (2001)

Αθήνα, Νοέμβριος 2008

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Διαχείριση Ρευμάτων Δεδομένων

Διδακτορική Διατριβή

του

Δημήτρη Σαχαρίδη

Διπλωματούχου Ηλεκτρολόγου Μηχανικού &
Μηχανικού Υπολογιστών Ε.Μ.Π. (2001)

Συμβουλευτική Επιτροπή: Τ. Σελλής
Ι. Βασιλείου
Α. Γ. Σταφυλοπάτης

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή τη $10^{η}$ Νοεμβρίου 2008.

Τ. Σελλής Καθ. ΕΜΠ	Ι. Βασιλείου Καθ. ΕΜΠ	Α. Γ. Σταφυλοπάτης Καθ. ΕΜΠ
Ε. Ζάχος Καθ. ΕΜΠ	Ν. Κοζύρης Αναπ. Καθ. ΕΜΠ	Σ. Σκιαδόπουλος Επ. Καθ. Παν. Πελ/σου
	Α. Δεληγιαννάκης Επ. Καθ. Πολ. Κρήτης	

Αθήνα, Νοέμβριος 2008

...

Δημήτρης Σαχαρίδης
Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Contents

List of Figures

List of Tables

PREFACE

This thesis is submitted in fulfilment of the requirements for the degree of Doctor of Philosophy, in the School of Electrical and Computer Engineering, National Technical University of Athens (NTUA), Greece. The presented work describes methods for managing data streams using approximation techniques and has been carried out the last four years in the Knowledge and Database Systems Laboratory (KDBSL) of NTUA.

Dimitris Sacharidis
Athens, November 2008

ABSTRACT

Driven by the enormous volumes of data communicated over today's Internet, several emerging data-management applications crucially depend on the ability to continuously generate, process, and analyze massive amounts of data in real time. A typical example domain here comprises the class of *continuous event-monitoring systems* deployed in a wide variety of settings, ranging from network-event tracking in large ISPs to transaction-log monitoring in large web-server farms and satellite-based environmental monitoring. For instance, tracking the operation of a nationwide ISP network requires monitoring detailed measurement data from thousands of network elements across several different layers of the network infrastructure. The volume of such monitoring data can easily become overwhelming (in the order of Terabytes per day). To deal effectively with the massive volume and continuous, high-speed nature of data in such environments, the *data streaming* paradigm has proven vital. Unlike conventional database query-processing engines that require several (expensive) passes over a static, archived data image, streaming data-analysis algorithms rely on building concise, approximate (but highly accurate) *synopses* of the input stream(s) in real-time (i.e., in one pass over the streaming data). Such synopses typically require space that is significantly sublinear in the size of the data and can be used to provide *approximate query answers* with guarantees on the quality of the approximation. In many monitoring scenarios, it is neither desirable nor necessary to maintain the data in full; instead, stream synopses can be used to retain enough information for the reliable reconstruction of the key features of the data required in analysis.

Chapter 1

Introduction

The need for management and data processing is becoming more intense due to the continuous technological development. In today's era, data are produced with constantly increasing volumes from increasingly more sources. Devices with logic circuits capable of processing and sensors collecting various measurements all acquire smaller size and require less energy. It is expected, therefore, that their use in everyday life will rapidly grow. Actually, we have reached the point where the amount of data is determined not by the rate of production from sources but from the rate of consumption by applications. The need for efficient management of this sea of data is becoming apparent.

The traditional model for managing and processing data, in general terms, involves the collection, modeling and storage in a database engine. The extraction of useful information and conclusions is made by posing queries based on some language. This model, which was the subject of the research community for many years, was developed and finalized more than three decades ago and is widely used today. Of course, its design is in accordance with specifications and requirements which are now considered outdated. In particular, we mention three main characteristics that distinguish the current requirements in relation to the traditional model. The first relates to the need for continuous processing and analysis of data as they are produced. The second with the huge volume of data generated. It is common that gigabytes of information per second are produced, which makes it difficult if not impossible to store them. Finally, note that many analysis applications are interested in the results rather than the queries themselves. For data related to such applications, the term *data streams* has been introduced to denote their dynamic nature: data stream from the sources of production, are not stored permanently and require immediate processing. Overviews of data-streaming issues and algorithms can be found, for instance, in [6, 72].

Furthermore, queries related to data streams have different characteristics. The principal characteristic is that they require continuous evaluation as new data stream. Whenever a new piece of information related to a query appears, the answer should be updated. The term *continuous queries* better describes this property. Typically in such questions only a recent segment, a *time window*, from the entire history of data evolution is interesting. The windows, besides their semantic role, are also used for practicality. Many continuous queries can not be evaluated in the entire history of streams and require the extraction of a smaller and easier manageable section.

It is evident that many of the assumptions made on conventional data bases do

not apply in the case of streams and hence new techniques and algorithms should be developed. A data streaming environment introduces resource restrictions to conventional static data processing algorithms, due to the high volumes and rates associated with incoming data. Namely: (i) there is not enough space to store the entire stream, as it can be of potentially unbounded size, and thus, data stream items can only be seen once; (ii) data stream items must be processed quickly in real time; and (iii) queries over data streams are of persistent nature and must be continuously and, most importantly, quickly evaluated. Under these restrictions, data stream processing algorithms must have small space requirements and exhibit fast per-item processing and querying time — here, small and quickly should be read as poly-logarithmic to data stream size.

Due to the above limitations, it is imperative to use summaries or *synopses* of streams which lead to an approximate evaluation of queries. This thesis deals with the construction of summarization structures and the development of fast algorithms that allow the management of very demanding data streams for various applications. In every case a key consideration is to provide as good as possible guarantees for the quality of results. In summary, the proposed methodology is discussed in relation to the following, often conflicting, axes.

Processing Time of a Streaming Tuple. For the processing of data streams various structures that reside in the main memory are used in order to maintain the system. These structures must be updated directly as new records reach the system.

Space for Storing Data Structures. Processing takes place entirely in the main memory. Since the main memory is an expensive commodity (2 to 3 orders of magnitude more expensive than secondary memory) and that each time there may be many multiple queries that compete for memory, it is desirable that the smallest possible amount of memory is used for the evaluation of continuous questions.

Query Answering Time. For each newly streaming record the system should assess the continuous query and decide if and how the result should change. This axis reflects the time required after processing an update is finished until the result is updated.

Communication Cost. In the case where multiple remote data streams exist, we should take into consideration the cost of communicating the records in the central server. The cost is usually measured by the size of information transferred rather than by the time required for communication.

Result Accuracy. The approximate evaluation always introduces an error in the query results. It is necessary to consider various metrics to assess how accurate is the result obtained in connection with the result that would be taken if there were available unlimited computing resources.

In conclusion, analyzing the factors that make data stream management an interesting problem, we discern the following.

- **Necessity** The current trend of research in the area of data management defends the evaluation of queries directly as data are produced. As many traditional queries cannot be evaluated on the model of streams, we have turned to approximation methods. As a result the need for new methods was born. Their main characteristic is the representation of the most important traits of the data in summarization structures while providing quality guarantees.

- **Interestingness** The shift towards approximation methods raises several interesting matters. For example, what is the interesting information that a stream of data carries and how do we find it in order to create summaries? Another matter is how much computational resources (memory and processing cycles) will be spent on constructing these summaries? Finally, we should define proper metrics estimating the quality of the synopses.

- **Feasibility** The methods proposed in this thesis can be implemented in existing data management systems. The reason is that summaries can be regarded as a simplification of data streams allowing direct in-place usage. They constitute the basis for the formulation and management of more complex queries in a transparent manner.

- **Challenging** The fact that the approach of this thesis is realistic, does not mean that there exist no difficult or challenging problems. A series of research challenges result from adapting the traditional model of data management into that of streams. The most important is that evaluation algorithms should operate in one pass over the streams and require sublinear space and time.

1.1 Contributions

This dissertation presents various methods for managing data streams using synopses. We focus on three data stream types, time series, update and moving objects streams. For the first two types we devise general-purpose structures such as *wavelet synopses* and *sketches* for summarizing them. Then, we turn our attention to streams produced by objects moving freely in space. We present techniques for creating and maintaining two distinct synopses, *spatial* and *spatiotemporal*. Our contributions include the following.

1. We consider conventional wavelet synopses for generic multidimensional data streams. For time series streams we introduce two novel operators, *SHIFT-SPLIT*, operating directly on summaries of wavelet transformed data. The operators allow for the management of streams so as an appropriate balance between the necessary space and time consuming is found. We provide the first concrete results for handling multidimensional data streams using these operations. Furthemore, we present three additional application fields that benefit from the SHIFT-SPLIT operations. The methods discussed and the results obtained appear in [50].

2. For update multidimensional data streams, we introduce a novel structure, termed *Group Count Sketch* (GCS), suitable for applications where the available computing resources are very limited. This sketch requires small space, features fast update and query times and offers a significantly large design freedom as it can be tailor-suited to the specific characteristics of the application. Based on GCS, we present a very efficient method for maintaining conventional wavelet synopses for update streams. The methods discussed and the results obtained appear in [17].

3. We introduce a novel indexing method for wavelet synopses, termed *Hierarchically Compressed Wavelet Synopses*, which consumes less space while offering

the same guarantees of accuracy. For finding the best summary under a space restriction we design a dynamic programming technique that optimally solves the problem. As its time complexity can be large, we also propose an approximation algorithm with tunable accuracy and a fast greedy algorithm. Even though the latter offers no accuracy guarantees it performs almost as effectively as the optimal and more importantly has time requirements on par with the conventional synopses algorithms. The methods discussed and the results obtained appear in [81].

4. We consider spatial synopses for moving objects data streams. In particular we consider the problem of maintaining *spatial k-medoid synopses*. We propose two techniques, based on Hilbert curves, for online maintenance in a distributed environment. We apply these techniques in a centralized setting, where moving objects continually push stream updates to the coordinator, and in a decentralized setting, where moving objects filter stream updates. In both settings, the main objective is to update very quickly and with low communication cost the k-medoid synopses. The methods discussed and the results obtained appear in [77].

5. In applications where the evolution of the objects' trajectories must be captured we introduce a novel type of *spatiotemporal synopses*, termed motion paths. These synopses examine the recent history and simplify trajectories by extracting common simplified routes. We propose techniques for a decentralized setting. Furthermore, our approach takes into account the ambiguity in the reported locations of the objects and approximates them with a given quality guarantee. The methods discussed and the results obtained appear in [82].

1.2 Outline

The remainder of this thesis is structured as follows.

Chapter 2 establishes the necessary background for introducing our proposed methodology. In particular, it introduces the types and models of data streams considered in this thesis and presents some necessary preliminaries regarding the wavelet decomposition.

Chapter 3 presents methods for constructing conventional wavelet synopses for data streams. For the time series model, the SHIFT-SPLIT operations are introduced and then applied for stream synopses as well as in other applications. For the update model, we describe the randomized data structure Group Count Sketch and elaborate on its application to wavelet synopses maintenance. We present an extensive experimental evaluation for both models.

Chapter 4 introduces the concept of hierarchically compressed wavelet synopses. Initially, we draw comparisons with conventional synopses and specify their shortcomings. Then, we present an optimal dynamic programming, an approximation and a greedy algorithm for the problem of maintaining a hierarchically compressed wavelet synopsis under the time series model. We present an extensive experimental evaluation for all algorithms.

Chapter 5 discusses synopses for moving objects streams. We discuss a framework for maintaining k-medoid synopses which summarize the current distribution

of objects in space. Then, we present a method for constructing spatiotemporal motion path synopses that summarize the recent history of the objects' trajectories. We present an extensive experimental evaluation for both types of synopses.

Chapter 6 concludes the discussion of this thesis summarizing its contributions. Finally, we identify possible extensions and propose future work.

Chapter 2

Preliminaries

In this chapter we provide the necessary background for understanding the methods introduced in the next chapters. In particular, Section 2.1 examines the schema of typical data streams and distinguishes among several types of data streams. Then, Section 2.2 establishes some preliminary notions regarding the wavelet decomposition used in Chapters 3 and 4 for constructing general purpose synopses.

2.1 Data Stream Types

A *data stream* is essentially a multiset (bag) containing multiple *streaming tuples*. A streaming tuple has the form $\langle timestamp, attr_1, attr_2, \ldots \rangle$, where $attr_1, attr_2, \ldots$ correspond to a regular relational attribute values and *timestamp* to a time measurement that marks the arrival of the tuple in the stream.

For the purpose of this thesis we distinguish three types of data streams that cover a broad range of applications.

2.1.1 Time Series Streams

We begin the discussion considering single dimensional streaming tuples of the form $\langle t_i, v_i \rangle$, where t_i is the timestamp and v_i is a value in a numerical domain V. Stream tuples appear ordered by t_i. As an example, consider an environmental sensor that continuously measures air temperature; v_i is the temperature at time t_i in this setting. Another way of viewing time series streams is by considering v as an infinitely long vector. As time progresses, the data stream *renders* v element by element, i.e., tuple $\langle t_i, v_i \rangle$ renders $v[t_i]$.

Multidimensional time series streams have the form $\langle t_i, a_j, b_k, \ldots, v_i \rangle$, where $a_j, b_k, \ldots$ correspond to other dimensions besides time t_i. Stream tuples appear ordered by t_i; however, multiple entries (with different $a_j, b_k, \ldots$ values) may appear at the same timestamp. Intuitively, a multidimensional data stream can be thought of as a multidimensional array v with dimensions $t_i, a_j, b_k, \ldots$. As time progresses, the data stream renders a particular cell of v, i.e, tuple $\langle t_i, a_j, b_k, \ldots, v_i \rangle$ renders $v[t_i, a_j, b_k, \ldots]$. Continuing the previous example, assume that there are multiple environmental sensors located at different areas. Then, streaming tuples have the form $\langle t_i, x_j, y_k, v_i \rangle$, where x_j, y_k identify the location of the reading.

Time series synopses must summarize the entire history of the stream, i.e., the entire array v, from a point in the past up to the current timestamp. The array

v quickly grows larger than the main memory due to its continuously increasing temporal dimension. However, note that v's projection at a particular timestamp, e.g., all measurements from all sensors at this timestamp, fits in main memory.

Popular synopses of this type are the wavelet synopses which are examined in Chapters 3 and 4.

2.1.2 Update Streams

Single dimensional update streams have the form $\langle t_i, a_j, \delta_i \rangle$, where t_i is the timestamp, a_j is a value in the dimension A and δ_i is the update (increment or decrement) for the a_j-th domain value. Intuitively, assume a vector v of length equal to A's domain size. The tuple $\langle t_i, a_j, \delta_i \rangle$ indicates that the value $v[a_j]$ is *updated* by δ_i, i.e., $v_{t_i}[a_j] \leftarrow v_{t_{i-1}}[a_j] + \delta_i$. As an example, consider a network monitoring application where v measures in bytes the traffic towards IP addresses. Then, t_i indicates the timestamp of a measurement specifying that IP a_j has received δ_j bytes. Note that in update streams, the time is not a dimension of the vector; it just marks the arrival of an update.

Multidimensional update streams have the form $\langle t_i, a_j, b_k, \ldots, \delta_i \rangle$. Consider a multidimensional array v with dimensions $a_j, b_k, \ldots$; a streaming tuple updates a single cell in v. In the previous example, assume that the application monitors traffic between a pair of IP addresses. Therefore, the stream tuples are $\langle t_i, a_j, b_k, \delta_i \rangle$, where t_i is the timestamp, a_j, b_k is the source and destination addresses and δ_i is the amount of data transferred.

Update synopses must summarize and maintain the current state of array v, i.e., after incorporating all updates up to now. The challenge in update streams is that v is too large to be maintained in main memory.

Popular synopses of this type are the wavelet synopses built on top of sketches which are examined in Chapter 3.

2.1.3 Moving Objects Streams

Moving objects streams have the form $\langle t_i, o_j, x_j, y_j \rangle$, where t_i is the timestamp, o_j is the object identifier and x_j, y_j are the current coordinates of object o_j at time t_i. Tuples with same timestamps are possible, as long as they have different object ids. The projection of such a stream on a single object o_j is called the *trajectory* of o_j and can be represented as a (monotone in t) polyline in the spatiotemporal xyt domain. A moving objects stream is essentially a collection of such polylines.

Depending on the application there are two categories as to which information is relevant and should be maintained. The former states that only the current position of all objects is relevant. On the other hand, the latter states that the entire evolution of all objects' trajectories is relevant. We refer to synopses that are suitable for applications of the former category as *spatial* and of the latter as *spatiotemporal* synopses.

A popular kind of spatial synopses are the k-medoids, whereas a novel spatiotemporal synopsis is the motion paths. Both synopses are investigated in detail in Chapter 5.

2.2 Introduction to the Wavelet Transformation

As discussed in Section 2.1, due to the infeasibility of storing the entire data stream, we need to resort to synopses approximating the entire history of a stream. For the first two types, i.e., time series and update streams, we consider a widely used summarization structure termed the *wavelet synopsis*. Chapters 3 and 4 discuss in detail algorithms for constructing good, in terms of accuracy, wavelet synopses. This section provides a brief introduction to the *discrete wavelet transformation* (DWT), which is the basis for wavelet synopses.

DWT is a mathematical tool for the hierarchical decomposition of functions, with a long history of successful applications in signal and image processing [51, 85]. Applying the wavelet transform to a (one- or multidimensional) data vector and retaining a small collection of the largest wavelet coefficient gives a very effective form of lossy data compression. Such wavelet synopses provide concise, general-purpose summaries of relational data, and can form the foundation for fast and accurate approximate query processing algorithms, such as approximate selectivity estimates, OLAP range aggregates and approximate join and multi-join queries. Wavelet synopses can also give accurate (one- or multidimensional) *histograms* of the underlying data vector at multiple levels of resolution, thus providing valuable primitives for effective data visualization.

The Discrete Wavelet Transformation creates "rough" and "smooth" views of the data at different resolutions. In the case of the Haar DWT that we use throughout this chapter, the "smooth" view consists of averages or average coefficients, whereas the "rough" view consists of differences or detail coefficients. At each resolution, termed level of decomposition or scale, the averages and details are constructed by pairwise averaging and differencing of the averages of the previous level. Consider the one-dimensional data vector $a = [2, 2, 0, 2, 3, 5, 4, 4]$ ($N = 8$). The Haar DWT of a is computed as follows. We first average the values together pairwise to get a new "lower-resolution" representation of the data with the pairwise averages $[\frac{2+2}{2}, \frac{0+2}{2}, \frac{3+5}{2}, \frac{4+4}{2}] = [2, 1, 4, 4]$. This averaging loses some of the information in a. To restore the original a values, we need *detail coefficients*, that capture the missing information. In the Haar DWT, these detail coefficients are the differences of the (second of the) averaged values from the computed pairwise average. Thus, in our simple example, for the first pair of averaged values, the detail coefficient is 0 since $\frac{2-2}{2} = 0$, for the second it is -1 since $\frac{0-2}{2} = -1$. No information is lost in this process — one can reconstruct the eight values of the original data array from the lower-resolution array containing the four averages and the four detail coefficients. We recursively apply this pairwise averaging and differencing process on the lower-resolution array of averages until we reach the overall average. The final Haar DWT of a is given by $\widehat{a} = [11/4, -5/4, 1/2, 0, 0, -1, -1, 0]$, that is, the overall average followed by the detail coefficients in order of increasing resolution. Each entry in $\widehat{a}$ is called a *wavelet coefficient*. The main advantage of using $\widehat{a}$ instead of the original data vector a is that for vectors containing similar values most of the detail coefficients tend to have very small values. Thus, eliminating such small coefficients from the wavelet transform (i.e., treating them as zeros) introduces only small errors when reconstructing the original data, resulting in a very effective form of lossy data compression [85].

We denote by $u_{j,k}$ and $w_{j,k}$ the k-th average (also called scaling coefficient) and

the k-th detail coefficient (also called wavelet coefficient), respectively, for the j-th level of decomposition. The averages at level j are decomposed into averages and details of level $j+1$. If we denote the set of scaling coefficients at the j-th level by U_j and the set of wavelet coefficients at the j-th level by W_j, we can formally write the previous statement as $U_j = U_{j+1} \oplus W_{j+1}$, where the direct-sum $\oplus$ notation refers to the decomposition process. The original data are the scaling coefficients of the 0-th level. For example, the 3 level decomposition of Figure 2.1 is decomposed as:

$$\begin{aligned} U_0 &= U_1 \oplus W_1 \\ &= U_2 \oplus W_2 \oplus W_1 \\ &= U_3 \oplus W_3 \oplus W_2 \oplus W_1 \ . \end{aligned}$$

Figure 2.1 also shows that for each level of decomposition j, there are 2^{n-j} wavelet coefficients $w_{j,k}$ and 2^{n-j} scaling coefficients, for $0 \leq k \leq 2^{n-j} - 1$. The transformed vector $\widehat{a}$ consists of the average, $u_{n,0}$, as its first element, followed by the details $w_{j,k}$ sorted decreasing by level j and increasing by position k: $u_{n,0}$, $w_{n,0}$, $w_{n-1,0}$, $w_{n-1,1}$, $w_{n-2,0}$, ..., $w_{1,0}$, ..., $w_{1,2^{n-1}-1}$.

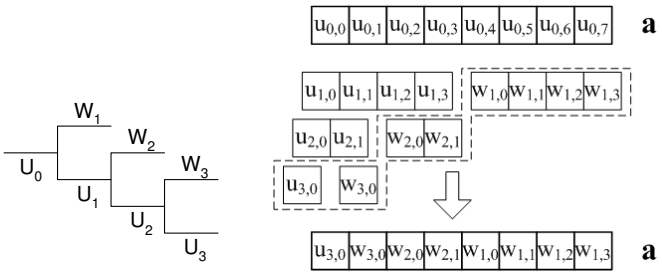


Figure 2.1: *Haar wavelet decomposition*

The *support interval* of a (wavelet or scaling) coefficient is the part of the original data that this coefficient depends on. Figure 2.2 shows the support intervals of Haar wavelets for a vector of size 8. A (wavelet or scaling) coefficient *covers* another (wavelet or scaling) coefficient if the support interval of the latter is (completely) contained in the support interval of the former. For example, the first coefficient in the second level of decomposition $w_{2,0}$ covers the first and second coefficients of the first level of decomposition, $w_{1,0}$ and $w_{1,1}$; see Figure 2.2. Haar wavelet coefficients $w_{j,k}$ and Haar scaling coefficients $u_{j,k}$ have the property that their support intervals are *dyadic intervals*, i.e, $[k2^j, (k+1)2^j - 1]$.

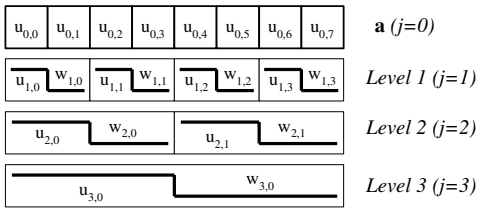


Figure 2.2: *Support intervals of Haar wavelets*

Intuitively, wavelet coefficients with larger support carry a higher weight in the reconstruction of the original data values. To equalize the importance of all Haar

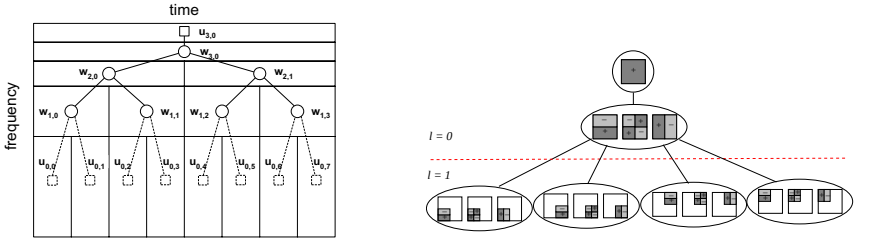
DWT coefficients, a common normalization scheme is to scale the coefficient values at level l (or, equivalently, the basis vectors $\phi_{l,k}$) by a factor of $\sqrt{N/2^l}$. This normalization essentially turns the Haar DWT basis vectors into an *orthonormal basis* — letting c_i^* denote the normalized coefficient values, this fact has two important consequences: (1) The *energy* of the a vector is preserved in the wavelet domain, that is, $\|a\|_2^2 = \sum_i a[i]^2 = \sum_i (c_i^*)^2$ (by Parseval's theorem); and, (2) Retaining the B largest coefficients in terms of *absolute normalized value* gives the (provably) best B-term wavelet synopsis in terms of Sum-Squared-Error (SSE) in the data reconstruction (for a given budget of coefficients B) [85].

Multidimensional Wavelet Transformation. There are two distinct ways to generalize the Haar DWT to the multidimensional case, the *standard* and *nonstandard* Haar decomposition [85]. Each method results from a natural generalization of the one-dimensional decomposition process described above, and both have been used in a wide variety of applications. Consider the case where a is a d-dimensional data array, comprising N^d entries. As in the one-dimensional case, the Haar DWT of a results in a d-dimensional wavelet-coefficient array $\widehat{a}$ with N^d coefficient entries. The non-standard Haar DWT works in $\log N$ phases where, in each phase, *one step* of pairwise averaging and differencing is performed across each of the d dimensions; the process is then repeated recursively (for the next phase) on the quadrant containing the averages across all dimensions. The standard Haar DWT works in d phases where, in each phase, a *complete* 1-dimensional DWT is performed for each one-dimensional row of array cells along dimension k, for all $k = 1, \ldots, d$. (full details and efficient decomposition algorithms are in [13, 88].) The supports of non-standard d-dimensional Haar coefficients are d-dimensional hyper-cubes (over dyadic ranges in $[N]^d$), since they combine 1-dimensional basis functions from the same resolution levels across all dimensions. The cross product of a standard d-dimensional coefficient (indexed by, say, $(i_1, \ldots, i_d)$) is, in general a d-dimensional hyper-rectangle, given by the cross-product of the 1-dimensional basis functions corresponding to coefficient indexes $i_1, \ldots, i_d$. Both multidimensional decompositions preserve the orthonormality, thus, retaining the largest B coefficient values gives an SSE-optimal B-term approximation of a.

Wavelet Tree. The multiresolution property of the Haar wavelets induces a tree construct capturing and illustrating this property. A wavelet coefficient w is the parent of another coefficient w', when w is the coefficient with the smallest support that covers w'. For Haar wavelets, which is our case, this tree is a binary tree where each node $w_{j,k}$ has exactly two children, $w_{j-1,2k}$ and $w_{j-1,2k+1}$. The scaling coefficient $u_{n,0}$ is the root of the tree having only one child $w_{n,0}$. This tree structure has been given several names in the wavelet bibliography, such as error tree [89, 88], dependency graph [84], etc. Figure 2.3(a) shows this tree for a vector of size 8; scaling coefficients are shown with squares, whereas wavelet coefficients are shown in circles. Figure 2.3(a) also shows the original data as children of the leaf nodes of the tree, drawn with dotted line. A nice property of this tree is that it portrays the way Haar wavelets partition the time-frequency plane; see Figure 2.3(a). As j decreases we gain accuracy in the time domain, but simultaneously, we lose accuracy in the frequency domain and vice versa.

Wavelet tree structures can be used to conceptualize the properties of both forms of d-dimensional Haar DWTs. In the non-standard case, the wavelet tree is essentially a quadtree (with a fanout of 2^d), where all internal non-root nodes contain 2^{d-1}

coefficients that have the same support region in the original data array but with different quadrant signs (and magnitudes) for their contribution. For standard d-dimensional Haar DWT, the wavelet tree structure is essentially a "cross-product" of d one-dimensional wavelet trees with the support and signs of coefficient $(i_1, \ldots, i_d)$ determined by the product of the component one-dimensional basis vectors (for $i_1, \ldots, d$). Figure 2.3(b) depicts a simple example wavelet tree structure for the non-standard Haar DWT of a 2-dimensional 4×4 data array.



(a) One-dimensional data array ($N = 8$)

(b) Non-standard two-dimensional Haar coefficients for a 4×4 data array

Figure 2.3: *Example wavelet tree structures*

The following lemma is a consequence of the time-frequency trade-off. A single point in time domain depends on those wavelet coefficients in the path to the root. As a result, a data value can be reconstructed in time proportional to the tree height and thus in time logarithmic to the vector size.

Lemma 2.2.1. *Let $\widehat{a}$ be the wavelet transform of vector a of size $N = 2^n$. Any value of a can be reconstructed using exactly $n + 1 = \log N + 1$ coefficients from $\widehat{a}$.*

Proof. Let $a[i]$ be the $(i+1)$-th value of a. At each level of decomposition j, there is exactly one wavelet coefficient $w_{j,\lfloor \frac{i}{2^j} \rfloor}$ that covers $a[i]$, because of the fact that Haar wavelets of the same level have non-overlapping support. This together with the parent-child relationship existent in the wavelet tree results in the covering wavelet coefficients $w_{j,\lfloor \frac{i}{2^j} \rfloor}$, $\forall j \in [1, n]$ and the scaling coefficient $u_{n,0}$ belonging to a $(n+1)$-long path in the tree. □

Therefore, as Lemma 2.2.1 suggests a point query can be answered using $O(\log N)$ coefficients. In the multidimensional case, updating a single data entry in the d-dimensional data array a impacts the values of $(2^d - 1) \log N + 1 = O(2^d \log N)$ coefficients in the non-standard case, and $(\log N + 1)^d = O(\log^d N)$ coefficients in the standard case.

The wavelet transformation is mainly used for its ability to answer range-sum queries also using $O(\log N)$ coefficients, as the following lemma suggests.

Lemma 2.2.2. *Let $\widehat{a}$ be the wavelet transform of vector a of size $N = 2^n$. A range-sum query $\sum_{i=l}^{r-1} a[i]$ can be answered using not more than $2n + 1 = 2 \log N + 1$ coefficients from $\widehat{a}$.*

Proof. This lemma holds because of the fact that Haar wavelets have a 0-th vanishing moment. For more details refer to [83]. □

In the multidimensional case, the bound becomes $O(2^d \log N)$ coefficients in the non-standard case, and $(\log N + 1)^d = O(\log^d N)$ coefficients in the standard case.

Recently there has been an increasing interest for wavelet in data-management applications. Matias et al. [64] first proposed the use of Haar-wavelet coefficients as synopses for accurately estimating the selectivities of range queries. Vitter and Wang [88] describe I/O-efficient algorithms for building multidimensional Haar wavelets from large relational datasets and show that a small set of wavelet coefficients can efficiently provide accurate approximate answers to range aggregates over OLAP cubes. Chakrabarti et al. [13] demonstrate the effectiveness of Haar wavelets as a general-purpose approximate query processing tool by designing efficient algorithms that can process complex relational queries (with joins, selections, etc.) entirely in the wavelet-coefficient domain. Schmidt and Shahabi [83] present techniques using the Daubechies family of wavelets to answer general polynomial range-aggregate queries. Deligiannakis and Roussopoulos [23] introduce algorithms for building wavelet synopses over data with multiple measures.

The work in [33] showed that it is possible to deterministically construct wavelet synopses for the same problem as in [32] and provided a novel dynamic programming recurrence, extensible [34] to any distributive error metric. Similar ideas were employed in [73] to construct optimal synopses in sub-quadratic time for a particular class of error metrics. Further, the work in [41] improves the space requirements of the aforementioned dynamic programming algorithms. For the same problem of optimal weighted synopses, the work in [62] constructs a wavelet-like basis so that the Parseval's theorem applies and, thus, the conventional greedy thresholding technique can be used. Assuming all range-sum queries are of equal importance, the authors in [63] proved that the heuristics employed in [64] are in fact optimal. The works in [42, 43] showed that for error metrics other than SSE, keeping the original coefficient values is suboptimal. Hence, they propose approximation algorithms for constructing *unrestricted* wavelet synopses that involve searching for the best value to assign for each coefficient stored.

More recently, wavelets have found broad use in data stream environments. The dynamic maintenance of Haar synopses was first studied in [65]. The works in [17, 38] use sketching techniques for maintaining conventional wavelet synopses over rapidly changing data streams.The approximation schemes of [42, 43] for unrestricted wavelet synopses are also extensible for the case of time-series data streams. A fast greedy algorithm for maximum-error metrics was introduced in [54] for the problem of constructing wavelet synopses over time-series data streams.

Due to its simplicity and nice properties (Lemmata 2.2.1 and 2.2.2), the wavelet decomposition is now broadly used as an alternative summarization technique to histograms for streaming data. However, several interesting research issues have been raised. In particular, there is a need for methods that offer space-time tradeoffs in maintaining wavelet synopses under very demanding time series and update streams. In rapid rate streams, it is convenient to occupy more memory in order to speed up the synopsis construction and update times. We study space-time tradeoffs in Chapter 3. Another issue that arises is the optimal use of allocated memory. We propose the hierarchical compression of wavelet synopses in Chapter 4.

Chapter 3

Conventional Wavelet Synopses

In this chapter we focus on constructing wavelet synopses for data streams. We consider the conventional method of indexing wavelet coefficients and use the standard *Sum-Squared Error* (SSE) for measuring the errors introduced. Two models of data streams are examined. For the time series model, we introduce the *SHIFT-SPLIT* operations that operate directly on the wavelet domain and offer tunable trade-offs. For the update model, we present a novel summary structure, termed *Group Count Sketch* (GCS), which allows for fast identification and extraction of the most important wavelet coefficients comprising the synopsis.

Some important wavelet transformation properties have not been fully explored and thus not exploited for the two common forms of multidimensional decomposition. In the following, we present in detail two novel operations, termed SHIFT and SPLIT, which work directly in the wavelet domain. We demonstrate their significance and usefulness by analytically proving two important results establishing a space-time trade-off for the summarization of time series multidimensional data streams. Furthermore we discuss the applicability of our methods in other settings including the transformation of massive datasets, archiving data and partial data reconstruction, which leads to significant I/O cost reduction in all cases.

In the case of the more demanding update data streams, randomized sketch synopses provide accurate approximations for wavelet synopses. The focus of existing work has typically been on minimizing *space requirements* of the maintained synopsis. However, to effectively support high-speed data-stream analysis, a crucial practical requirement is to also optimize: the update time for incorporating a streaming data element in the sketch, and the query time for producing a good synopsis from the sketch. Such time costs must be small enough to cope with rapid stream-arrival rates and the real-time querying requirements of typical streaming applications. With cheap and plentiful memory, space is often only a secondary concern after query/update time costs. We propose the first fast solution to the problem of tracking wavelet representations of one-dimensional and multi-dimensional data streams, based on a novel stream synopsis, the Group Count Sketch (GCS).

The remainder of this chapter is organized as follows. Section 3.1 motivates our approached for maintaining conventional wavelet synopses on data streams and reviews relevant bibliography. Section 3.4 introduces a disk block allocation strategy for wavelet coefficients. Section 3.2 presents the SHIFT and SPLIT operations. Then, Section 3.3 discusses the main results for time series streams. Section 3.5 includes extensions of the SHIFT-SPLIT operations to other applications. Section 3.6

establishes the necessary background on handling update streams. Section 3.7 presents the group-count sketch and Section 3.8 applies it for update streams. An extensive experimental evaluation is included in Section 3.9. The chapter concludes with Section 3.10.

3.1 Motivation and Related Work

Despite its broad acceptance, the wavelet transformation has not been explored to its full potential for data intensive applications. In particular, the compact support and the multi-scale properties of the wavelets, as illustrated by the wavelet tree of decomposition, lead to some overlooked but interesting properties. With the exception of [13], where traditional relational algebra operations are re-defined to work directly in the wavelet domain, most applications resort to reconstruction of many data values to support even the simplest operations in the original domain. We introduce two general-purpose operations for wavelet decomposed data, named SHIFT and SPLIT, that stem from the multiresolution properties of wavelets to provide general purpose functionality. They are designed to work directly in the wavelet domain and can be utilized in a wide range of data intensive applications, resulting in significant improvements in every case. Here we apply them for the time series model of data streams.

Gilbert et al. [38] demonstrated that obtaining a best K-term wavelet approximation of a single dimensional time series stream of total size N is possible using space of $O(K+\log N)$ while the per stream item update cost is $O(\log N)$. In the following, we show that the SHIFT-SPLIT operations can further reduce the per-item cost to $O\left(\frac{1}{B}\log\frac{N}{B}\right)$ at the expense of additional storage of B coefficients. We also investigate the case of multidimensional data streams, decomposed under two different forms of wavelet transformation. We conclude that we can maintain a K-term approximation, under certain restrictions. To the best of our knowledge, this is the first work dealing with wavelet approximation of multidimensional data streams, as previous works [76, 10, 32, 38] focused on the single dimensional case.

Besides data streams, the SHIFT-SPLIT operations are suitable for other wavelet maintenance tasks. In particular, there is a strong dependency among wavelet coefficients, due to their multiresolution nature. This observation leads to assigning wavelet coefficients that are related with each other under a common access pattern to the same multidimensional tile. These tiles are then stored directly into the secondary storage, as their size is adjusted to fit a disk block. By using this tiling approach we can minimize the number of disk I/Os needed to perform any operation in the wavelet domain, including the important reconstruction operation which results in significant query cost reductions. We designed the SHIFT and SPLIT operations to work with multidimensional tiles, as these operations benefit significantly from their existence.

In update streams we need to resort to approximation techniques, known as *sketches*, for constructing synopses. We propose the first known streaming algorithms for space- and time-efficient tracking of approximate wavelet summaries for both one- and multidimensional data streams. Our approach relies on a novel, sketch-based stream synopsis structure, termed the *Group Count Sketch (GCS)* that allows us to provide similar space/accuracy tradeoffs as the simple sketches of [40], while guaranteeing: (1) small, logarithmic update times (essentially touch-

ing only a small fraction of the GCS for each streaming update) with simple, fast, hash functions; and, (2) polylogarithmic query times for computing the top wavelet coefficients from the GCS. In brief, our GCS algorithms rely on two key, novel technical ideas. First, we work *entirely in the wavelet domain*, in the sense that we directly sketch *wavelet coefficients*, rather than the original data vector, as updates arrive. Second, our GCSs employ *group structures based on hashing and hierarchical decomposition* over the wavelet domain to enable fast updates and efficient binary-search-like techniques for identifying the top wavelet coefficients in sublinear time. We also demonstrate that, by varying the degree of our search procedure, we can effectively explore the tradeoff between update and query costs in our GCS synopses. Our GCS algorithms and results also naturally extend to both the standard and non-standard form of the *multidimensional* wavelet transform, essentially providing the only known efficient solution for streaming wavelets in more than one dimension. As our experimental results with both synthetic and real-life data demonstrate, GCS synopses allow very fast update and searching, capable of supporting very high speed data sources.

Sketches first appeared for estimating the second frequency moment of a set of elements [2] and have since proven to be a useful summary structure in such a dynamic setting. Their application includes uses for estimating join sizes of queries over streams [1, 25], maintaining wavelet synopses [40], constructing histograms [37, 87], estimating frequent items [14, 19] and quantiles [39]. The work of Gilbert et al. [40] for estimating the most significant wavelet coefficients is closely related to ours. As we discuss, the limitation is the high query time required for returning the approximate representation. In follow-up work, the authors proposed a more theoretical approach with somewhat improved worst case query times [37]. This work considers an approach based on a complex construction of range-summable random variables to build sketches from which wavelet coefficients can be obtained. The update times remain large. Our bounds, discussed in Section 3.8, improve those that follow from [37], and our algorithm is much simpler to implement. In similar spirit, Thaper et al. [87] use AMS sketches to construct an optimal B-bucket histogram of large multidimensional data. No efficient search techniques are used apart from an exhaustive greedy heuristic which always chooses the next best bucket to include in the histogram; still, this requires an exhaustive search over a huge space. The idea of using *group-testing* techniques to more efficiently find heavy items appears in several prior works [19, 20, 37]; here, we show that it is possible to apply similar ideas to groups under L_2 norm, which has not been explored previously. Recently, different techniques have been proposed for constructing wavelet synopses that minimize non-Euclidean error metrics, under the time-series model of streams [42, 54].

The contributions of our work can be summarized as follows.

1. We introduce the SHIFT-SPLIT operations that work directly on transformed data and can expedite common maintenance tasks.

2. We propose a methodology based on SHIFT-SPLIT that offers space-time tradeoffs for maintaining time series wavelet synopses.

3. We apply the SHIFT-SPLIT operations obtaining IO efficient methods for three common wavelet-related tasks, i.e., transformation, appending and reconstruction.

4. We introduce the Group Count Sketch for summarizing update streams.

5. We provide theoretical results regarding the complexities of using the GCS for extracting wavelet synopses for both forms of multidimensional transformations.

6. We present extensive experimental results of our algorithms on both synthetic and real-life datasets. Our experimental study demonstrates: (i) the applicability of SHIFT-SPLIT operation for time-series streams; (ii) the improvement over current state-of-the-art transformation algorithms; (iii) the suitability of GCS based wavelet synopses for very demanding update streams.

3.2 Shift-Split Operations

In this section we introduce two general purpose operations, SHIFT and SPLIT, for wavelet transformed data. We start with the single dimensional case and then move to multiple dimensions.

3.2.1 Single Dimensional Shift-Split

There is a relationship among the coefficients in the transform of a vector, a and in the transform of a dyadic region b of the vector. This relationship is captured by *shifting*, re-indexing, the wavelet coefficients (details) of b and by *splitting*, calculating contributions from the scaling coefficient (average).

The SHIFT-SPLIT operations are better understood in the context of wavelet trees. Let a be a vector of size $N = 2^n$ and let b be the $(k+1)$-th dyadic range of vector a with size $M = 2^m$. The wavelet coefficients of $\widehat{a}$ are denoted by $w^a_{j,l}$, whereas the wavelet coefficients of $\widehat{b}$ are denoted by $w^b_{j,l}$; similarly for scaling coefficients, $u^a_{j,l}$ and $u^b_{j,l}$. Also, let T_a and T_b denote the wavelet trees of $\widehat{a}$ and $\widehat{b}$, respectively. Figure 3.1 illustrates the above.

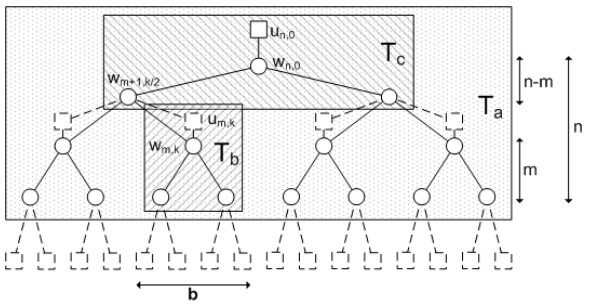


Figure 3.1: *Shift-Split operations*

The support of the wavelet coefficient $w^a_{m,k}$ is the dyadic range that b represents. Therefore, $w^a_{m,k}$ covers $w^b_{m,0}$ and vice versa, since their support is the same range of a; see T_b in Figure 3.1. Furthermore, all children of $w^a_{m,k}$ in T_a have common support with the corresponding children of $w^b_{m,0}$ in T_b. Specifically, at the j-th level of decomposition, the i-th coefficient $w^b_{j,i}$ of T_b has the same support with the $(k2^{m-j}+i)$-th coefficient $w^a_{j,k2^{m-j}+i}$.

Definition of SHIFT. Let a be a vector of size $N = 2^n$ and let b be the $(k+1)$-th dyadic range of vector a with size $M = 2^m$. Also, let $f : \mathbb{Z} \to \mathbb{Z}$, $f(i) = \left(\frac{k}{2^{m-j}} + i\right)$, be a function that translates the indices i of $\widehat{b}$ to indices $f(i)$ of $\widehat{a}$. The SHIFT operation on the transformed vector $\widehat{b}$ is defined as the re-indexing of the wavelet coefficients by function f. □

The wavelet coefficients of $\widehat{a}$ that cover the interval represented by b contain a portion of the energy of the average of vector b. To be exact, the value of the wavelet coefficients $w^a_{j,\lfloor \frac{k}{2^{j-m}} \rfloor}$ for $j \in [m+1, n]$, as well as the average $u^a_{n,0}$ depend on the value of the average $u^b_{m,0}$; these coefficients lie in the path from $w^a_{m,k}$ to the root and are contained in T_c of Figure 3.1. Essentially the value of the average $u^b_{m,0}$ is *split* across these $n - m + 1$ coefficients, contributing either positively, or negatively.

Definition of SPLIT. Let a be a vector of size $N = 2^n$ and let b be the $(k+1)$-th dyadic range of vector a with size $M = 2^m$. Also, let $g : [m+1, n] \to \mathbb{R}$,

$$g(j) = \begin{cases} \frac{1}{\sqrt{2}^{j-m}} u^b_{m,k}, & \text{if } k \mod 2^{j-m} \text{ even} \\ -\frac{1}{\sqrt{2}^{j-m}} u^b_{m,k}, & \text{if } k \mod 2^{j-m} \text{ odd} \end{cases}$$

be the function that calculates the contribution of $u^b_{m,k}$ per level j. The SPLIT operation on the transformed vector $\widehat{b}$ calculates the contribution of $u^b_{m,k}$ to the $n - m$ wavelet coefficients: $\delta w^a_{j,\lfloor \frac{k}{2^{j-m}} \rfloor} = g(j)$ for $j \in [m+1, n]$ and to the average: $\delta u^a_{n,0} = \frac{1}{\sqrt{2}^{n-m}} u^b_{m,k}$. □

To demonstrate the use of the SHIFT-SPLIT operations, let us look at two examples.

Example 3.2.1. *Assume we are to transform a very large vector a of size $N = 2^n$ into the wavelet domain, where only the subregion $[k2^m, (k+1)2^m - 1]$ of the vector contains non-zero values. Let b be that non-zero subregion of size $M = 2^m$. Because of the fact that b forms a dyadic interval, we can apply the SHIFT-SPLIT operations to construct $\widehat{a}$ as follows. First, we obtain the wavelet transform $\widehat{b}$ in time $O(M)$. Next, we apply the SHIFT operation to place the wavelet coefficients of $\widehat{b}$ in their corresponding position in $\widehat{a}$. Finally, we apply the SPLIT operation on the average of b to obtain $n - m + 1$ contributions and construct the remaining $n - m + 1$ coefficients. We have completed the wavelet transformation of a in time $O(M + n - m) = O(M + \log \frac{N}{M})$, instead of $O(N)$.* □

Example 3.2.2. *Assume we have already transformed vector a of size $N = 2^n$ into the wavelet domain. There are updates, stored in vector b, coming for a subregion $[k2^m, (k+1)2^m - 1]$ of a. The goal is to update the wavelet transform of a as efficiently as possible. Each of $|b| = M = 2^m$ updates requires $n+1$ values to be updated, leading to a total cost of $O(M \log N)$. However, we can use the SHIFT-SPLIT operations to batch updates and reduce cost, as follows. First, we obtain the wavelet transform $\widehat{b}$ in time $O(M)$. Next, we apply the SHIFT operation to calculate the indices of the wavelet coefficients of $\widehat{a}$ which need to be updated by the wavelet coefficients of $\widehat{b}$. Finally, we apply the SPLIT operation on the average of b to obtain $n - m + 1$ contributions and update the corresponding coefficients in $\widehat{a}$. The total update cost using SHIFT-SPLIT has been reduced to $O(M + \log \frac{N}{M})$.* □

	SHIFT	SPLIT
Standard	$O\left(\lceil\frac{M}{B}\rceil^d\right)$	$O\left(\left(\lceil\frac{M}{B}\rceil - \lceil\log_B \frac{N}{M}\rceil\right)^d - \lceil\frac{M}{B}\rceil^d\right)$
Non-Standard	$O\left(\lceil\frac{M}{B}\rceil^d\right)$	$O\left((2^d-1)\lceil\log_B \frac{N}{M}\rceil\right)$

Table 3.1: *Shift-Split of tiles*

3.2.2 Multidimensional Shift-Split

The SHIFT-SPLIT operations in the multidimensional decomposition exploit the relationship between the wavelet coefficients of the entire dataset and those in a multidimensional dyadic range. A multidimensional dyadic range is formed by the cross product of single dimensional dyadic intervals. For the non-standard decomposition we will only consider *cubic* multidimensional dyadic ranges resulting from dyadic intervals of equal length for all dimensions; arbitrary multidimensional dyadic ranges can always be seen as a collection of cubic intervals.

To perform the SHIFT-SPLIT operations for the standard multidimensional decomposition, one has to perform the operations for each dimension separately. Any coefficient in the d-dimensional dyadic interval can only be shifted or split in each dimension, and thus can sustain d operations in total. Consider as an example a d-dimensional dataset, where each dimension has size $N = 2^n$, and a cubic dyadic range of edge $M = 2^m$. The SHIFT operation affects $(M-1)^d$ coefficients and the SPLIT operation calculates $(M+n-m)^d - (M-1)^d$ contributions.

With the non-standard multidimensional transformation, all the wavelet coefficients in the cubic dyadic range must be shifted similar to the standard transformation. However, only the scaling coefficient has to be split and the contributions for the coefficients inside nodes on the path to the root have to be calculated. Therefore, in the non-standard transformation, the SHIFT operation affects $M^d - 1$ coefficients and the SPLIT operation calculates $(2^d-1)(n-m)+1$ contributions.

3.3 Wavelet Synopses for Time Series Streams

In this section, we present a methodology for maintaining a wavelet synopsis of a multidimensional data stream in the time-series model. The focus here is to construct a space and time efficient algorithm for maintaining the best K-term synopsis. We show that we cannot, in general, maintain a K-term synopsis for multidimensional datasets decomposed using the standard form under bounded space. However, if certain conditions are met we can maintain a K-term synopsis effectively.

Let us start with the simple one dimensional case. As shown in [38], we can maintain the best K-term approximation of a data of length $N = 2^n$ by using space $K + \log N + 1$. We always store the K highest coefficients encountered so far, plus those coefficients whose value can change by subsequent data arrivals. These coefficients, termed wavelet crest in [76], lie on the path from the current value to the root of the wavelet tree and therefore, they are exactly $\log N + 1$. Equivalently, if we consider a range containing just the data values under consideration, the SPLIT operation results in contributions lying in the wavelet crest. Therefore, at any time we have to keep the coefficients that can be affected by the SPLIT operation in memory.

Result 1. *A K-term wavelet synopsis of a data stream of size N in the time series model can be maintained using memory of $O(K + B + \log \frac{N}{B})$ coefficients with $O\left(\frac{1}{B} \log \frac{N}{B}\right)$ per-item computational cost.*

Proof. If we keep in memory a buffer of size $B = 2^b$ we can reduce per-item processing time at the expense of extra space. We collect B coefficients in the buffer, transform them and apply the SHIFT operation to obtain the $B-1$ relocated wavelet coefficients. Next, we compare these coefficients with the K highest, to obtain the new set of K highest coefficients. Finally, we have to update the coefficients that can change by using the contributions derived from the SPLIT operation. The number of contributions for a buffer of size B is $\log \frac{N}{B}$ and thus the space required for the coefficients on the crest is $\log \frac{N}{B}$. The total computational cost for the buffer, which includes the cost for transformation and the cost for updating the coefficients on the crest, is $O\left(B + \log \frac{N}{B}\right)$. As a result, the per-item computational cost is $O\left(\frac{1}{B} \log \frac{N}{B}\right)$ reduced from $O(\log N)$, at the expense of extra space of B. □

The key for being able to maintain a wavelet approximation in the one dimensional case is the fact that only a single path to the root of the wavelet tree has to be maintained at any time. Let us turn our attention to the multidimensional case. We assume that the data needs to be appended in only one dimension (usually the time dimension), which is the case for multidimensional data streams of the time series model. To separate the continuously increasing dimension, we let T denote its current size, whereas the other dimensions have a constant size of N. Therefore, the d-dimensional data stream has a size of $N^{d-1}T$. The amount of space, besides the K terms, required to maintain a K-term approximation depends on the number of coefficient that can be affected by a SPLIT operation. We calculate the number of these coefficients for each of the multidimensional forms, assuming that we have extra storage to buffer M^d coefficients, where $M = 2^m$.

Result 2. *A K-term standard wavelet synopsis of a d-dimensional data stream growing in the T dimension can be maintained using memory of $O\left(K + M^d + N^{d-1} \log \frac{T}{M}\right)$ coefficients.*

Proof. In the standard form, there are $d - 1$ wavelet trees of size N and a single wavelet tree of size T. Since, the stream expands on the dimension of size T, we only have to keep a path to the root for the wavelet tree corresponding to that dimension. However, a new data value can arrive in any position on the other trees, which means that we have to keep all the paths to the root for the $d-1$ trees. To recap, we need to keep all N 1-d basis functions from the $d - 1$ trees of size N and only $\log \frac{T}{M}$ 1-d basis functions for the tree corresponding to the dimension which increases. The cross product between these sets of 1-d basis functions results in $N^{d-1} \log \frac{T}{M}$ d-dimensional basis functions and thus that many coefficients have to be maintained, besides the K highest coefficients and the extra storage space of M^d coefficients used for buffering. Therefore, the required space of $O\left(K + M^d + N^{d-1} \log \frac{T}{M}\right)$ coefficients is prohibitive, except in the case where the constant dimensions have very small domain size, so that N^{d-1} is small. □

Result 3. *A K-term non-standard wavelet synopsis of a d-dimensional data stream growing in the T dimension can be maintained using memory of $O\left(K + M^d + (2^d - 1) \log \frac{N}{M} + \log \frac{T}{N}\right)$ coefficients.*

Proof. Since the dimension with size T is constantly expanding, we have to deal with non equal dimension sizes, similar to [13]. Such a data stream can be seen as a $\frac{T}{N}$ hypercubes of size N^d, where each of these hypercubes can be decomposed with the non-standard form. Each of these $\frac{T}{N}$ hypercubes results in a wavelet tree capturing the non-standard decomposition, where there exists a single average as the root of each of these trees. We apply the single dimensional transformation on the $\frac{T}{N}$ data constructed by these averages. The final result consists of $\frac{T}{N}$ non-standard multidimensional trees and a single one dimensional tree which has as leaf nodes the averages of the non-standard trees. We assume the z-ordered access pattern, described in Section 3.5.1, and we allow for extra buffering space of M^d coefficients. Under these restrictions, the coefficients we have to retain lie in a path to the root in the last tree of the hypercubes, and in the path to the root in the single dimensional wavelet tree. Therefore we need to keep $(2^d - 1) \log \frac{N}{M}$ coefficients from the non-standard tree and $\log \frac{T}{N}$ coefficients from the 1-d tree, resulting in a total space cost of $O\left(K + M^d + (2^d - 1) \log \frac{N}{M} + \log \frac{T}{N}\right)$. □

3.4 Storing Wavelets

Before we present additional wavelet applications we need to discuss the assignment of wavelet coefficients to disk blocks. We have already seen that the wavelet tree captures the dependency among coefficients. In particular, if a coefficient is required to be retrieved then all coefficients on the path to the root must also be retrieved. This property creates an access pattern of wavelets that must be exploited by the disk block allocation strategy.

Intuitively, a disk block should contain coefficients with overlapping support intervals, so that the utilization of the in-block coefficients is high. However, we must take under consideration the fact that the disk block allocation strategy should not allow redundancy, in that a wavelet coefficient should belong to one block only. Under this restriction, in order to be fair across all coefficients, we partition the wavelet tree into binary subtree tiles and store each tile on a disk block. Assuming that the disk block size B is a power of 2, $B = 2^b$, we achieve logarithmic utilization of the blocks. At least b coefficients inside the block, lying in a path, are to be utilized any time this disk block is needed. Logarithmic utilization may seem low at first, but it is the best we can hope for under our restrictions, as proven in [84].

One final issue is that the size of the binary subtree tiles is $2^b - 1$, whereas the block size is 2^b. We are wasting space of 1 coefficient in our block allocation strategy. Therefore, we choose to store the scaling coefficient corresponding to the root of the subtree, along with the wavelet coefficients of the tile. The extra scaling coefficients that we store are useful for query answering, as they can dramatically reduce query costs. An example of the disk block allocation strategy for a wavelet tree of 32 coefficients is shown in Figure 3.2.

In the standard multidimensional transformation each dimension is decomposed independently. Therefore, there cannot be a single tree capturing the levels of decomposition. In case of 2-d, considering a 1-d wavelet tree for each of the decomposed dimensions, two 1-d wavelet trees are required. Every coefficient in a transformed 2-d array has two indices, one for each dimension. Each of these indices identifies a position in the 1-d tree, which as we have seen corresponds to a decomposition level and to a translation inside that level. Figure 3.3 shows a coefficient in an 8×8 2-d

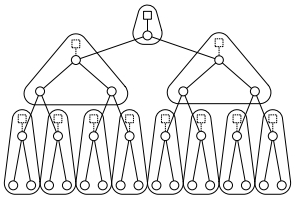


Figure 3.2: *Disk block allocation strategy*

array and the corresponding indices on the two wavelet trees.

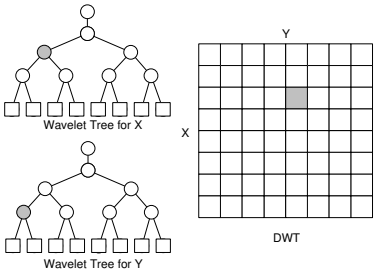


Figure 3.3: *Standard form wavelet trees*

The two 1-d trees can be used to determine which coefficients need to be retrieved for reconstructing data values on the 2-d array. Subsequently, they provide information about the access pattern of 2-d wavelets. A single data value on the untransformed (original) 2-d array corresponds to a path in each of the 1-d wavelet trees, or better, a set of 1-d indices, as mentioned before. The cross product among all indices across these sets, construct the 2-d indices whose coefficients must be retrieved. For a $N \times N$ array, where $N = 2^n$, each of the paths contains $(n + 1)$ 1-d indices, therefore there are $(n + 1)^2$ 2-d indices. Figure 3.4 shows the two paths on the 1-d wavelet trees, as well as the required coefficient resulting from the cross product between 1-d indices.

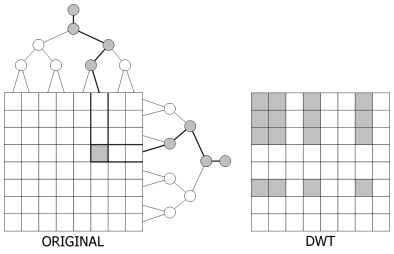


Figure 3.4: *Standard form data point reconstruction*

On the other hand, a single wavelet tree can capture the levels of decomposition and dependency among coefficients for the non-standard transformation. The support intervals of the wavelet coefficients form a quad-tree, as each support interval is further decomposed in quadrants at the next level of decomposition. At the j-th level of d-dimensional decomposition we have $(2^d)^j$ nodes, each containing $2^d - 1$ coefficients with support interval hypercubes with edge length 2^j.

In the 2-d case, the support intervals of the coefficients are squares with side length of power 2. There are 3 coefficients for each support interval, one for each of

the wavelet subspaces: W^d, W^v and W^h; thus, each quad tree node contains its 3 corresponding coefficients. Figure 3.5 shows the wavelet tree for an 8×8 array and zooms in on a multidimensional tile. The support interval of the children nodes, which are the four quadrants of the support interval of the parent node, are shown in dark grey. To reconstruct a point in the original 2-d array, one has to traverse the quad tree bottom up and use all 3 coefficients in each node.

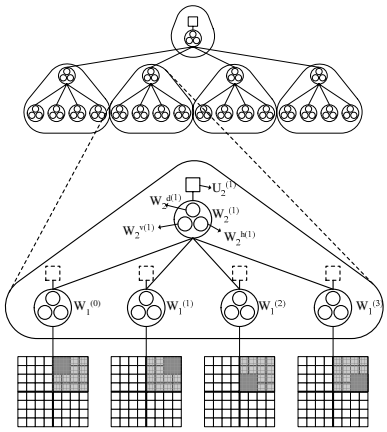


Figure 3.5: *Non-standard form wavelet tree*

As in the single dimensional case, our main concern is to pack coefficients in disk blocks so that we achieve the highest possible block utilization on query time and thus decreasing retrieval cost. The solution is to assign as many coefficients with the same support to the same disk block as possible. This results in different disk block allocation strategies for the two multidimensional forms of decomposition. We assume d-dimensional dataset, where each dimension has size $N = 2^n$. Furthermore, disk block size is B^d, where $B = 2^b$.

In the standard multidimensional decomposition, each dimension can be treated independently. Therefore, for each dimension we construct tiles of size B containing the B coefficients of a subtree, similar to the single dimensional case. The cross product of these d sets of single dimensional bases construct B^d multidimensional bases. The coefficients corresponding to these B^d bases are stored in the same block and form a multidimensional tile.

In the non-standard multidimensional decomposition, tiles are subtrees of the quad tree. The branching factor of a d-dimensional quad tree is $D = 2^d$ and each node contains $D - 1$ coefficients. Therefore, a tile of height b contains $\frac{D^b-1}{D-1}$ nodes or equivalently $D^b - 1$ coefficients. By also storing the scaling corresponding to the root node we create tiles of $D^b = (2^d)^b = (2^b)^d = B^d$ coefficients which fit in a disk block of size B^d. Figure 3.5 shows the tiling of a 8×8 array, for disk blocks of size 16.

SHIFT-SPLIT on disk blocks. In the following we extend the SHIFT-SPLIT operation to disk-resident data. We start with the single dimensional case of a vector of size $N = 2^n$ and its $k + 1$-th dyadic interval of size $M = 2^m$, when the disk block size is $B = 2^b$. The coefficients affected by the SHIFT operation belong to a subtree of the wavelet tree, and that subtree contains exactly $\lceil \frac{M}{B} \rceil$ tiles. On the other hand, the SPLIT operation calculates $\log \frac{N}{M}$ contributions. Because these contributions lie on a single path to the root inside every tile, there are $\log B$

coefficients affected per tile. This results in exactly $\lceil \log_B \frac{N}{M} \rceil$ tiles containing the contributions of the SPLIT operation. To summarize for the single dimensional case, the SHIFT operation affects B times less tiles than coefficients, whereas the SPLIT operation affects $\log B$ times less tiles than coefficients.

Extending to d-dimensional tiles of size $B^d = (2^b)^d$ and applying the observation for the single dimensional case, we derive the number of d-dimensional tiles affected by the operations in each multidimensional form. The results are summarized in Table 3.1. For the remainder we will drop the ceiling operations to increase readability.

3.5 Shift-Split Applications

Besides the construction of wavelet synopses, the SHIFT-SPLIT operations can prove useful in a variety of applications described in the following.

3.5.1 Efficient Transformation

One of the most important application of the SHIFT-SPLIT operations is I/O efficient transformation of massive multidimensional datasets. In the following, we assume that the dataset is d-dimensional with each dimension having a domain of size $N = 2^n$, so that the hypercube has N^d cells. The available memory for performing the transformation is M^d, where $M = 2^m$, measured in units of coefficients. Therefore, at any point in time, there can only be $M^d \ll N^d$ coefficients in main memory. Given these restrictions we need to construct an efficient, in terms of required I/O operations, algorithm for decomposing the dataset. We begin by assuming that one I/O operation involves a single data value, or coefficient. Later, we measure I/O operations in units of disk blocks, as we consider the optimal disk block allocation strategy described in Section 3.4.

The intuition behind our approach is simple. We assume that the data are either organized and stored in multidimensional chunks of equal size and shape, or that the chunk-organization process has been performed, similar to [13, 88]. We transform each chunk and use the SHIFT operation to relocate the coefficients and the SPLIT operation to update the stored coefficients. The chunks are hypercubes of size M^d so that they fit in main memory. Figure 3.6 shows a one dimensional example, for $N = 16$ and $M = 4$, where the current chunk is C. The transformation of C results in the wavelet coefficients inside the box needing to be shifted. The scaling coefficient of C must be split to calculate the contributions to the coefficients shown in grey. With black are shown the coefficients that have a finalized value; that is, coefficients that will not be affected by C or by chunks coming after C. With white are shown the coefficients that do not cover any of the chunks seen so far.

Result 4. *The I/O complexity for transforming a d - dimensional dataset with each dimension having domain size $N = 2^n$ into the standard form of decomposition using memory of M^d coefficients is $O\left(\left(\frac{N}{B} + \frac{N}{M}\log_B \frac{N}{M}\right)^d\right)$ disk blocks of size B^d.*

Proof. As mentioned in Section 3.2.2, the SHIFT operation, for the standard decomposition, affects $(M-1)^d$ coefficients, whereas the SPLIT operation affects $(M+n-m)^d-(M-1)^d$ coefficients. Consequently, each chunk requires $\$O\left((M+n-m)^d\right) =$

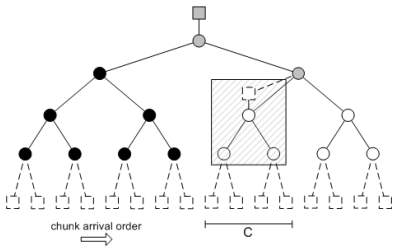


Figure 3.6: *Transformation by chunks*

$O\left((M+\log\frac{N}{M})^d\right)$ I/O operations. Summing for all $\left(\frac{N}{M}\right)^d$ chunks, we derive the I/O complexity, measured in terms of coefficients, for the standard multidimensional wavelet transformation: $O\left(\left(N+\frac{N}{M}\log\frac{N}{M}\right)^d\right)$. Now, let us consider disk blocks of size B^d, for $B=2^b$. In this case, the I/O cost per chunk in units of disk blocks is: $O\left(\left(\frac{M}{B}+\log_B\frac{N}{M}\right)^d\right)$. Summing for all chunks we derive the I/O complexity, measured in terms of disk blocks, for the standard multidimensional wavelet transformation: $O\left(\left(\frac{N}{B}+\frac{N}{M}\log_B\frac{N}{M}\right)^d\right)$ □

Vitter et al. [89, 88] use the standard form to decompose multidimensional datasets, without taking under consideration, however, our optimal block allocation strategy. They transform a dense d-dimensional dataset in $O(N_z^d\log_M N)$disk I/O operations; in the case of sparse data with N_z non-zero values the I/O complexity is $O(N_z^d\log_M N)$. We can modify our SHIFT-SPLIT approach to accommodate for sparseness similar to the latter case, where only N_z non-zero values exist; the modified I/O complexity is $O\left(\left(N_z+\frac{N_z}{M}\log\frac{N}{M}\right)^d\right)$. However, for comparison purposes we omit the effect of sparseness in the original data. The I/O complexities are summarized in Table 3.2.

Result 5. *The I/O complexity for transforming a d - dimensional dataset with each dimension having domain size $N=2^n$ into the non-standard form of decomposition using memory of $M^d+(2^d-1)\log\frac{N}{M}$ coefficients is $O\left(\left(\frac{N}{B}\right)^d\right)$ disk blocks of size B^d.*

Proof. In the case of the non-standard multidimensional wavelet transformation, the SHIFT operation affects M^d-1 coefficients, whereas the SPLIT operation affects $(D-1)(n-m)+1$coefficients, where $D=2^d$. The per chunk I/O cost is $O\left(M^d+(D-1)\log\frac{N}{M}\right)$. Summing for all $\left(\frac{N}{M}\right)^d$chunks, we derive the I/O complexity, measured in terms of coefficients, for the non-standard multidimensional wavelet transformation: $O\left(N^d+(D-1)\left(\frac{N}{M}\right)^d\log\frac{N}{M}\right)$. When tiling is used, the I/O cost per chunk in units of disk blocks becomes: $O\left(\left(\frac{M}{B}\right)^d+(D-1)\log_B\frac{N}{M}\right)$. Summing for all chunks we derive the I/O complexity, measured in terms of disk blocks, for the non-standard multidimensional wavelet transformation: $O\left(\left(\frac{N}{B}\right)^d+(D-1)\left(\frac{N}{M}\right)^d\log_B\frac{N}{M}\right)$. However, if we enforce a particular access pattern on the chunks, namely a z-ordering, and allow some extra amount of memory $(2^d-1)\log\frac{N}{M}$ to store those coefficients that are affected by the splitting of the scaling coefficient of the chunks, we can reduce the cost to the optimal $O(N^d)$, as seen in Table 3.2. A similar approach has been suggested in [13], where a recursive procedure is used to ensure values come in the particular access pattern. □

Transformation Method	I/O cost *(in coefficients)*	I/O cost *(in blocks)*
Vitter et al. (Standard)	$O\left(N^d \log_M N\right)$	
Shift-Split (Standard)	$O\left(\left(N + \frac{N}{M}\log\frac{N}{M}\right)^d\right)$	$O\left(\left(\frac{N}{B} + \frac{N}{M}\log_B\frac{N}{M}\right)^d\right)$
Shift-Split (Non-Standard)	$O\left(N^d\right)$	$O\left(\left(\frac{N}{B}\right)^d\right)$

Table 3.2: *I/O complexities*

3.5.2 Appending

In this section we investigate the problem of appending new data to existing transformed data. Appending is fundamentally different from updating in that it results in the increase of the domain of one or more dimensions. As a result, the wavelet decomposed dimensions also grow, new levels of transformation are introduced and therefore the transform itself changes. We would like to perform appending directly in the wavelet domain, preserving as much of the transformed data as possible and avoiding reconstruction of the original data. The SHIFT-SPLIT operations helps us achieve this goal. To make complexity analysis easier, we omit the effect of the optimal disk block allocation strategy, or equivalently assume disk block size of 1 coefficient. Also, we use the standard form of decomposition, as analysis for the non-standard form is similar.

As a motivation, consider the scenario where a massive multidimensional dataset containing measurements over 10 years is decomposed into the wavelet domain to expedite query processing. A new set of data for the following year has become available, which results in appending to the time domain and possibly on other measure dimensions. Let us assume that the 10-year decomposed d-dimensional dataset has size of N^d, and that the available memory is M^d, for $N = 2^n$ and $M = 2^m$.

Our SHIFT-SPLIT approach to the problem is the following, repeating for each M^d data values that we gather in memory. We start by performing the d-dimensional DWT on the gathered data. Next, if required, we make the necessary space on the original transformed data (expand) to accommodate for the new data to be appended. The final step is to shift and split the gathered data to update the expanded data. The second step is the most important in the appending application. Let us assume that we must expand on one of the dimensions to accommodate for the coefficients held in memory. The expansion means that the wavelet tree for that dimension has to increase its height by 1, and thus double its domain range. This expansion process is carried out by shifting and splitting the decomposed data in this dimension. Figure 3.7 shows expansion in one dimension, where T_{old} becomes T_{new} and $|T_{new}| = 2|T_{old}|$. The expansion step creates the necessary space for the current chunk of M^d coefficients in memory, as well as for some of the next chunks. Therefore, this step, although costly, is rather rare.

The I/O cost of expanding transformed data in one dimension is $O(N^d)$ as all coefficients have to be shifted to construct the new data cube of size $2N^d$. Note, that although the asymptotic cost is high, the required SHIFT-SPLIT operations are very fast, which leads to fast execution times for expanding the domain of one dimension. This phenomenon is amplified by the use of tiling and is demonstrated in Section 3.9.1. Moreover, this operation, unlike reconstruction, does not require memory to process. The I/O cost of applying the SHIFT-SPLIT operations on the

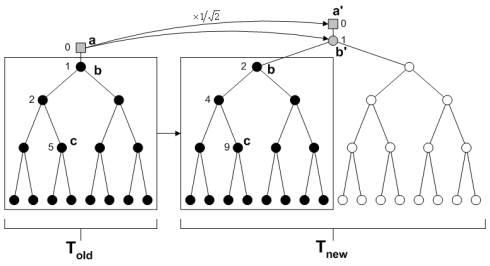


Figure 3.7: *Wavelet tree expanding*

memory chunk of size M^d is $O\left(\left(M + \log \frac{N}{M}\right)^d\right)$.

3.5.3 Partial Reconstruction

In this section, we discuss the problem of reconstructing a set of values specified by a range on a multidimensional dataset. The problem is equivalent to translating the selection operation of relational algebra to the wavelet domain. Chakrabarti et al. [13] have provided a solution for the non-standard form, in which they identify the coefficients who cover the range and calculate their contribution. Here, we present a similar approach, based on the inverse of SHIFT-SPLIT operations, which generalizes to both forms of decomposition. The inverse of SHIFT is essentially the inverse index translation, whereas the inverse of SPLIT is Lemma 2.2.1, which shows how to reconstruct a value from contributions on a path to the root. Therefore, the cost of the inverses of these operations is the same.

We focus our discussion here to multidimensional ranges that are dyadic ranges; an arbitrary selection range can be seen as a number of such dyadic ranges. Therefore, our problem degenerates to the reconstruction of a d-dimensional dyadic range of size M^d, given the transformation of the entire data of size N^d. The scaling coefficients of the dyadic range are calculated using the inverse SHIFT, whereas the rest of the coefficients are simply calculated from the coefficients in the original dataset by re-indexing, using the inverse SPLIT.

Result 6. *The time complexity for reconstructing a d-dimensional dyadic range of size M^d from a wavelet transformed signal of size N^d is $O\left(\left(M + \log \frac{N}{M}\right)^d\right)$ for the standard form and $O\left(M^d + (D-1)\log \frac{N}{M}\right)$ for the non-standard.*

Proof. It follows from the complexity of the SHIFT-SPLIT operations. □

3.6 Dealing with Update Streams

In this section, we first discuss the basic elements of our stream-processing model and briefly introduce AMS sketches [2].

Our input comprises a continuous stream of update operations, rendering a data vector a of N values (i.e., the data-domain size). Without loss of generality, we assume that the index of our data vector takes values in the integer domain $[N] = \{0, \ldots, N-1\}$, where N is a power of 2 (to simplify the notation). Each streaming update is a pair of the form $(i, \pm v)$, denoting a net change of $\pm v$ in the $a[i]$ entry; that is, the effect of the update is to set $a[i] \leftarrow a[i] \pm v$. Intuitively, "$+v$" ("$-v$") can

be seen as v insertions (resp., deletions) of the i^{th} vector element, but more generally we allow entries to take negative values, similar to [72]. For d data dimensions, a is a d-dimensional vector (*tensor*) and each update $(i_1, \ldots, i_d, \pm v)$ effects a net change of $\pm v$ on entry $a[i_1, \ldots, i_d]$. Without loss of generality we assume a domain of $[N]^d$ for the d-dimensional case — different dimension sizes can be handled in a straightforward manner. Further, our methods do not need to know the domain size N beforehand — standard adaptive techniques can be used.

In the data-streaming context, updates are only seen *once in the (fixed) order of arrival*; furthermore, the rapid data-arrival rates and large data-domain size N make it impossible to store a explicitly. Instead, our algorithms can only maintain a concise *synopsis* of the stream that requires only sublinear space, and, at the same time, can (a) be maintained in small, sublinear processing time per update, and (b) provide query answers in sublinear time. Sublinear here means polylogarithmic in N, the data-vector size. (More strongly, our techniques guarantee update times that are sublinear in the *size of the synopsis*.)

The randomized *AMS sketch* [2] is a broadly applicable stream synopsis structure based on maintaining randomized linear projections of the streaming input data vector a. Briefly, an *atomic AMS sketch* of a is simply the *inner product* $\langle a, \xi \rangle = \sum_i a[i]\xi(i)$, where ξ denotes a random vector of four-wise independent ± 1-valued random variates. Such variates can be easily generated on-line through standard pseudo-random hash functions $\xi()$ using only $O(\log N)$ space (for seeding) [2, 40]. To maintain this inner product over the stream of updates to a, initialize a running counter X to 0 and set $X \leftarrow X \pm v\xi(i)$ whenever the update $(i, \pm v)$ is seen in the input stream. An *AMS sketch* of a comprises several independent atomic AMS sketches (i.e., randomized counters), each with a different random hash function $\xi()$. The following theorem summarizes the key property of AMS sketches for stream-query estimation, where $||v||_2$ denotes the L_2-norm of a vector v, so $||v||_2 = \sqrt{\langle v, v \rangle} = \sqrt{\sum_i v[i]^2}$.

Theorem 3.6.1 ([1, 2])**.** *Consider two (possibly streaming) data vectors a and b, and let Z denote the $O(\log(1/\delta))$-wise median of $O(1/\epsilon^2)$-wise means of independent copies of the atomic AMS sketch product $(\sum_i a[i]\xi_j(i))(\sum_i b[i]\xi_j(i))$. Then, $|Z - \langle a, b \rangle| \leq \epsilon ||a||_2 ||b||_2$ with probability $\geq 1 - \delta$.*

Thus, using AMS sketches comprising only $O(\frac{\log(1/\delta)}{\epsilon^2})$ atomic counters we can approximate the vector inner product $\langle a, b \rangle$ to within $\pm\epsilon ||a||_2 ||b||_2$ (hence implying an ϵ-relative error estimate for $||a||_2^2$).

3.7 The Group-Count Sketch

We introduce a novel, hash-based probabilistic synopsis data structure, termed *Group-Count Sketch (GCS)*, that can estimate the energy (squared L_2 norm) of fixed groups of elements from a vector $\widehat{a}$ of size N under our streaming model. (To simplify the exposition we initially focus on the one-dimensional case, and present the generalization to multiple dimensions later in this section.) Our GCS synopsis requires small, sublinear space and takes sublinear time to process each stream update item; more importantly, we can use a GCS to obtain a high-probability estimate of the energy of a group within additive error $\epsilon \|\widehat{a}\|_2^2$ in *sublinear time*. We

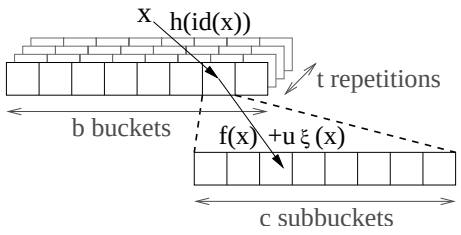


Figure 3.8: *Our Group-Count Sketch (GCS) data structure: x is hashed (t times) to a bucket and then to a subbucket within the bucket, where a counter is updated.*

then demonstrate how to use GCSs as the basis of efficient streaming procedures for tracking large wavelet coefficients.

Our approach takes inspiration from the AMS sketching solution for vector L_2-norm estimation; still, we need a much stronger result, namely the ability to estimate L_2 norms for a (potentially large) number of *groups of items* forming a partition of the data domain $[N]$. A simple solution would be to keep an AMS sketch of each group separately; however, there can be *many* groups, linear in N, and we cannot afford to devote this much space to the problem. We must also process streaming updates as quickly as possible. Our solution is to maintain a structure that first partitions items of $\widehat{a}$ into their group, and then maps groups to buckets using a hash function. Within each bucket, we apply a second stage of hashing of items to sub-buckets, each containing an atomic AMS sketch counter, in order to estimate the L_2 norm of the bucket. In our analysis, we show that this approach allows us to provide accurate estimates of the energy of any group in $\widehat{a}$ with tight $\pm\epsilon\|\widehat{a}\|_2^2$ error guarantees.

Assume a total of k groups of elements of $\widehat{a}$ that form a partition of $[N]$. For notational convenience, we use a function `id` that identifies the specific group that an element belongs to, $\texttt{id} : [N] \rightarrow [k]$. (In our setting, groups correspond to fixed dyadic ranges over $[N]$ so the `id` mapping is trivial.) Following common data-streaming practice, we first define a basic randomized estimator for the energy of a group, and prove that it returns a good estimate (i.e., within $\pm\epsilon\|\widehat{a}\|_2^2$ additive error) with constant probability $> \frac{1}{2}$; then, by taking the median estimate over t independent repetitions, we are able to reduce the probability of a bad estimate to exponentially small in t. Our basic estimator first hashes groups into b buckets and then, within each bucket, it hashes into c sub-buckets. (The values of t, b, and c parameters are determined in our analysis.) Furthermore, as in AMS sketching, each item has a $\{\pm 1\}$ random variable associated with it. Thus, our GCS synopsis requires three sets of t hash functions, $h_m : [k] \rightarrow [b]$, $f_m : [N] \rightarrow [c]$, and $\xi_m : [N] \rightarrow \{\pm 1\}$ ($m = 1, \ldots, t$). The randomization requirement is that h_m's and f_m's are drawn from families of pairwise independent functions, while ξ_m's are four-wise independent (as in basic AMS); such hash functions are easy to implement, and require only $O(\log N)$ bits to store.

Our GCS synopsis s consists of $t \cdot b \cdot c$ counters (i.e., atomic AMS sketches), labeled $s[1][1][1]$ through $s[t][b][c]$, that are maintained and queried as follows:

UPDATE(i, u). Set $s[m][h_m(\texttt{id}(i))][f_m(i)] += u \cdot \xi_m(i)$, for each $m = 1, \ldots, t$.

ESTIMATE(GROUP). Return the estimate $\text{median}_{m=1,\ldots,t} \sum_{j=1}^{c} (s[m][h_m(\text{GROUP})][j])^2$ for the energy of the group of items GROUP $\in \{1, \ldots, k\}$ (denoted by $\|\text{GROUP}\|_2^2$).

Thus, the update and query times for a GCS synopsis are simply $O(t)$ and $O(t \cdot c)$,

respectively. The following theorem summarizes our key result for GCS synopses.

Theorem 3.7.1. *Our Group-Count Sketch algorithms estimate the energy of item groups of the vector $\widehat{a}$ within additive error $\epsilon\|\widehat{a}\|_2^2$ with probability $\geq 1-\delta$ using space of $O\left(\frac{1}{\epsilon^3}\log\frac{1}{\delta}\right)$ counters, per-item update time of $O\left(\log\frac{1}{\delta}\right)$, and query time of $O\left(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\right)$.*

Proof. Fix a particular group GROUP and a row r in the GCS; we drop the row index m in the context where it is understood. Let BUCKET be the set of elements that hash into the same bucket as GROUP does: $\text{BUCKET} = \{i \mid i \in [1,n] \wedge h(\mathtt{id}(i)) = h(\text{GROUP})\}$. Among those, let COLL be the set of elements other than those of GROUP: $\text{COLL} = \{i \mid i \in [1,n] \wedge \mathtt{id}(i) \neq \text{GROUP} \wedge h(\mathtt{id}(i)) = h(\text{GROUP})\}$. In the following, we abuse notation in that we refer to a refer to both a group and the set of items in the group with the same name. Also, we write $\|S\|_2^2$ to denote the sum of squares of the elements (i.e. L_2^2) in set S: $\|S\|_2^2 = \sum_{i\in S}\widehat{a}[i]^2$.

Let *est* be the estimator for the sum of squares of the items of GROUP. That is, $est = \sum_{j=1}^{c} est_j$ where $est_j = (s[m][h_m(\text{GROUP})][j])^2$ is the square of the count in sub-bucket SUB_j. The expectation of this estimator is, by simple calculation, the sum of squares of items in sub-bucket j, which is a fraction of the sum of squares of the bucket. Similarly, using linearity of expectation and the four-wise independence of the ξ hash functions, the variance of *est* is bounded in terms of the square of the expectation:

$$\mathsf{E}[est] = \mathsf{E}[\|\text{BUCKET}\|_2^2] \qquad\qquad \mathsf{Var}[est] \leq \tfrac{2}{c}\mathsf{E}[\|\text{BUCKET}\|_2^4]$$

To calculate $\mathsf{E}[\|\text{BUCKET}\|_2^2]$, observe that the bucket contains items of GROUP as well as items from other groups denoted by the set COLL which is determined by h. Because of the pairwise independence of h, this expectation is bounded by a fraction of the total energy. Therefore:

$$\begin{aligned}
\mathsf{E}[\|\text{BUCKET}\|_2^2] &= \|\text{GROUP}\|_2^2 + E[\|\text{COLL}\|_2^2] \leq \|\text{GROUP}\|_2^2 + \tfrac{1}{b}\|\widehat{a}\|_2^2 \\
\text{and } \mathsf{E}[\|\text{BUCKET}\|_2^4] &= \|\text{GROUP}\|_2^4 + \mathsf{E}[\|\text{COLL}\|_2^4] + 2\|\text{GROUP}\|_2^2\mathsf{E}[\|\text{COLL}\|_2^2] \\
&\leq \|\widehat{a}\|_2^4 + \tfrac{1}{b}\|\widehat{a}\|_2^4 + 2\|\widehat{a}\|_2^2\cdot\tfrac{1}{b}\|\widehat{a}\|_2^2 \leq (1+\tfrac{3}{b})\|\widehat{a}\|_2^4 \leq 2\|\widehat{a}\|_2^2
\end{aligned}$$

since $\|\text{GROUP}\|_2^2 \leq \|\widehat{a}\|_2^2$ and $b \geq 3$. The estimator's expectation and variance satisfy

$$\mathsf{E}[est] \leq \|\text{GROUP}\|_2^2 + \tfrac{1}{b}\|\widehat{a}\|_2^2 \qquad\qquad \mathsf{Var}[est] \leq \tfrac{4}{c}\|\widehat{a}\|_2^4$$

Applying the Chebyshev inequality we obtain $\mathsf{Pr}\left[|est - \mathsf{E}[est]| \geq \lambda\|\widehat{a}\|_2^2\right] \leq \dfrac{4}{c\lambda^2}$ and by setting $c = \frac{32}{\lambda^2}$ the bound becomes $\frac{1}{8}$, for some parameter λ. Using the above bounds on variance and expectation and the fact that $|x-y| \geq ||x|-|y||$ we have,

$$|est - \mathsf{E}[est]| \geq \left|est - \|\text{GROUP}\|_2^2 - \frac{1}{b}\|\widehat{a}\|_2^2\right| \geq \left|\left|est - \|\text{GROUP}\|_2^2\right| - \frac{1}{b}\|\widehat{a}\|_2^2\right|.$$

Consequently (note that $\mathsf{Pr}[|x| > y] \geq \mathsf{Pr}[x > y]$),

$$\mathsf{Pr}\left[\left|est - \|\text{GROUP}\|_2^2\right| - \frac{1}{b}\|\widehat{a}\|_2^2 \geq \lambda\|\widehat{a}\|_2^2\right] \leq \mathsf{Pr}\left[|est - \mathsf{E}[est]| \geq \lambda\|\widehat{a}\|_2^2\right] \leq \frac{1}{8}$$

or equivalently, $\mathsf{Pr}\left[|est - \|\text{GROUP}\|_2^2| \geq \left(\lambda + \frac{1}{b}\right)\|\widehat{a}\|_2^2\right] \leq \frac{1}{8}$. Setting $b = \frac{1}{\lambda}$ we get $\mathsf{Pr}\left[|est - \|\text{GROUP}\|_2^2| \geq 2\lambda\|\widehat{a}\|_2^2\right] \leq \frac{1}{8}$ and to obtain an estimator with $\epsilon\|\widehat{a}\|_2^2$ additive error we require $\lambda = \frac{\epsilon}{2}$ which translates to $b = O(\frac{1}{\epsilon})$ and $c = O(\frac{1}{\epsilon^2})$.

By Chernoff bounds, the probability that the median of t independent instances of the estimator deviates by more than $\epsilon\|\widehat{a}\|_2^2$ is less than e^{-qt}, for some constant q. Setting this to the probability of failure δ, we require $t = O\left(\log\frac{1}{\delta}\right)$, which gives the claimed bounds.

□

3.8 Wavelet Synopses for Update Streams

In the following we present a solution for maintaining wavelet synopses under the update model of data streams, using the group-count sketches. Our goal is to continuously track a compact B-coefficient wavelet synopsis under our general, high-speed update-stream model. We require our solution to satisfy all three key requirements for streaming algorithms outlined earlier in this chapter, namely: (1) sublinear synopsis space, (2) sublinear per-item update time, and (3) sublinear query time, where sublinear means polylogarithmic in the domain size N. As in [40], our algorithms return only an *approximate* synopsis comprising (at most) B Haar coefficients that is provably near-optimal (in terms of the captured energy of the underlying vector) assuming that our vector satisfies the *"small-B property"* (i.e., most of its energy is concentrated in a small number of Haar DWT coefficients) — this assumption is typically satisfied for most real-life data distributions [40].

The streaming algorithm presented by Gilbert et al. [40] (termed "GKMS" in the following) focuses primarily on the one-dimensional case. The key idea is to maintain an AMS sketch for the streaming data vector a (as discussed in Section 3.6). To produce the approximate B-term representation, GKMS employs the constructed sketch of a to estimate the inner product of a with all wavelet basis vectors, essentially performing an exhaustive search over the space of all wavelet coefficients to identify important ones. Although techniques based on range-summable random variables constructed using Reed-Muller codes were proposed to reduce or amortize the cost of this exhaustive search by allowing the sketches of basis vectors to be computed more quickly, the overall query time for discovering the top coefficients remains superlinear in N (i.e., at least $\Omega(\frac{1}{\epsilon^2} N \log N)$), violating our third requirement. For large data domains, say $N = 2^{32} \approx 4$ billion (such as the IP address domain considered in [40]), a query can take a very long time: over an hour, even if a million coefficient queries can be answered per second! This essentially renders a direct extension of the GKMS technique to multiple dimensions infeasible since it implies an exponential explosion in query cost (requiring at least $O(N^d)$ time to cycle through all coefficients in d dimensions). In addition, the update cost of the GKMS algorithm is *linear in the size of the sketch* since the whole data structure must be "touched" for each update. This is problematic for high-speed data streams and/or even moderate sized sketch synopses.

Our Approach. Our proposed solution relies on two key novel ideas to avoid the shortcomings of the GKMS technique. First, we work *entirely in the wavelet domain*: instead of sketching the original data entries, our algorithms sketch the wavelet-coefficient vector $\widehat{a}$ as updates arrive. This avoids any need for complex range-summable hash functions. Second, we employ *hash-based grouping* in conjunction with *efficient binary-search-like techniques* to enable very fast updates as well as identification of important coefficients in polylogarithmic time.

– *Sketching in the Wavelet Domain.* Our first technical idea relies on the observation that we can efficiently produce sketch synopses of the stream *directly in the wavelet domain*. That is, we translate the impact of each streaming update on the relevant wavelet coefficients. By the linearity properties of the DWT and our earlier description, we know that an update to the data entries corresponds to only polylogarithmically many coefficients in the wavelet domain. Thus, on receiving an update to a, our algorithms directly convert it to $O(\text{polylog}(N))$ updates to the wavelet

coefficients, and maintain an approximate representation of the wavelet coefficient vector $\widehat{a}$.

– *Time-Efficient Updates and Large-Coefficient Searches.* Sketching in the wavelet domain means that, at query time, we have an approximate representation of the wavelet-coefficient vector $\widehat{a}$ and need to be able to identify all those coefficients that are "large", relative to the total energy of the data $\|\widehat{a}\|_2^2 = \|a\|_2^2$. While AMS sketches can give us these estimates (a point query is just a special case of an inner product), querying remains much too slow taking at least $\Omega(\frac{1}{\epsilon^2}N)$ time to find which of the N coefficients are the B largest. Note that although a lot of earlier work has given efficient streaming algorithms for identifying high-frequency items [14, 19, 61], our requirements here are quite different. Our techniques must monitor items (i.e., DWT coefficients) whose values increase and decrease over time, and which may very well be *negative* (even if all the data entries in a are positive). Existing work on "heavy-hitter" tracking focuses solely on non-negative frequency counts [19] often assumed to be non-decreasing over time [14, 61]. More strongly, we must find items whose *squared value* is a large fraction of the total vector energy $\|\widehat{a}\|_2^2$: this is a stronger condition since such "L_2^2 heavy hitters" may not be heavy hitters under the conventional sum-of-counts definition. [1]

At a high level, our algorithms rely on a *divide-and-conquer* or *binary-search-like* approach for finding the large coefficients. To implement this, we need the ability to efficiently estimate sums-of-squares for *groups* of coefficients, corresponding to dyadic subranges of the domain $[N]$. We then disregard low-energy regions and recurse only on high-energy groups — note that this guarantees no false negatives, as a group that contains a high-energy coefficient will also have high energy as a whole. Furthermore, our algorithms also employ *randomized, hash-based grouping* of dyadic groups and coefficients to guarantee that each update only touches a small portion of our synopsis, thus guaranteeing very fast update times.

3.8.1 Hierarchical Search Structure

We apply our GCS synopsis and estimators to the problem of finding items with large energy (i.e., squared value) in the $\widehat{a}$ vector. Since our GCS works in the wavelet domain (i.e., sketches the wavelet coefficient vector), this is exactly the problem of recovering important coefficients. To efficiently recover large-energy items, we impose a regular tree structure on top of the data domain $[N]$, such that every node has the same degree r. Each level in the tree induces a partition of the nodes into groups corresponding to *r-adic ranges*, defined by the nodes at that level. [2] For instance, a binary tree creates groups corresponding to dyadic ranges of size 1, 2, 4, 8, and so on. The basic idea is to perform a search over the tree for those high-energy items above a specified energy threshold, $\phi\|\widehat{a}\|_2^2$. We can prune groups with energy below the threshold and, thus, avoid looking inside those groups: if the estimated energy is accurate, then these cannot contain any high-energy elements. Our key result is that, using such a hierarchical search structure of GCSs, we can provably (within appropriate probability bounds) retrieve all items above the threshold plus a controllable error quantity ($(\phi+\epsilon)\|\widehat{a}\|_2^2$), and retrieve no elements below the threshold

[1] For example, consider a set of items with counts $\{4, 1, 1, 1, 1, 1, 1, 1, 1\}$. The item with count 4 represents $\frac{2}{3}$ of the sum of the squared counts, but only $\frac{1}{3}$ of the sum of counts.

[2] Thus, the `id` function for level l is easily defined as $\mathtt{id}_l(i) = \lfloor i/r^l \rfloor$.

minus that small error quantity ($(\phi - \epsilon)\|\widehat{a}\|_2^2$).

Theorem 3.8.1. *Given a vector wwidehata of size N we can report, with high probability $\geq 1-\delta$, all elements with energy above $(\phi+\epsilon)\|\widehat{a}\|_2^2$ (where $\phi \geq \epsilon$) within additive error of $\epsilon\|\widehat{a}\|_2^2$ (and therefore, report no item with energy below $(\phi-\epsilon)\|\widehat{a}\|_2^2$) using space of $O\left(\frac{\log_r N}{\epsilon^3} \cdot \log \frac{r \log_r N}{\phi\delta}\right)$, per item processing time of $O\left(\log_r N \cdot \log \frac{r \log_r N}{\phi\delta}\right)$ and query time of $O\left(\frac{r}{\phi\epsilon^2} \cdot \log_r N \cdot \log \frac{r \log_r N}{\phi\delta}\right)$.*

Proof. Construct $\log_r N$ GCSs (with parameters to be determined), one for each level of our r-ary search-tree structure. We refer to an element that has energy above $\phi\|\widehat{a}\|_2^2$ as a "hot element", and similarly groups that have energy above $\phi\|\widehat{a}\|_2^2$ as "hot ranges". The key observation is that all r-adic ranges that contain a hot element are also hot. Therefore, at each level (starting with the root level), we identify hot r-adic ranges by examining only those r-adic ranges that are contained in hot ranges of the previous level. Since there can be at most $\frac{1}{\phi}$ hot elements, we only have to examine at most $\frac{1}{\phi}\log_r N$ ranges and pose that many queries. Thus, we require the failure probability to be $\frac{\log_r N}{\phi\delta}$ for each query so that, by the union bound, we obtain a failure probability of at most δ for reporting all hot elements. Further, we require each level to be accurate within $\epsilon\|\widehat{a}\|_2^2$ so that we obtain all hot elements above $(\phi + \epsilon)\|\widehat{a}\|_2^2$ and none below $(\phi - \epsilon)\|\widehat{a}\|_2^2$. The theorem follows. □

Setting the value of r gives a tradeoff between query time and update time. Asymptotically, we see that the update time decreases as the degree of the tree structure, r, increases. This becomes more pronounced in practice, since it usually suffices to set t, the number of tests, to a small constant. Under this simplification, the update cost essentially reduces to $O(\log_r N)$, and the query time reduces to $O(\frac{r}{\epsilon^2\phi}\log_r N)$. (We will see this clearly in our experimental analysis.) The extreme settings of r are 2 and N: $r = 2$ imposes a binary tree over the domain, and gives the fastest query time but $O(\log_2 N)$ time per update; $r = N$ means updates are effectively constant $O(1)$ time, but querying requires probing the whole domain, a total of N tests to the sketch.

3.8.2 Sketching in the Wavelet Domain

As discussed earlier, given an input update stream for data entries in a, our algorithms build GCS synopses on the corresponding wavelet coefficient vector $\widehat{a}$, and then employ these GCSs to quickly recover a (provably good) approximate B-term wavelet representation of a. To accomplish the first step, we need an efficient way of "translating" updates in the original data domain to the domain of wavelet coefficients (for both one- and multidimensional data streams).

– *One-Dimensional Updates.* An update (i, v) on a translates to the following collection of $\log N + 1$ updates to wavelet coefficients (that lie on the path to leaf $a[i]$):

$$\left(0, 2^{-\frac{1}{2}\log N} v\right), \left\{\left(2^{\log N - l} + k, (-1)^{k \mod 2} 2^{-\frac{l}{2}} v\right) : \text{ for each } l = 0, \ldots, \log N - 1\right\},$$

where $l = 0, \ldots, \log N - 1$ indexes the resolution level, and $k = \lfloor i 2^{-l} \rfloor$. Note that each coefficient update in the above set is easily computed in constant time.

– *multidimensional Updates.* We can use exactly the same reasoning as above to produce a collection of (constant-time) wavelet-coefficient updates for a given data

update in d dimensions. As explained, the size of this collection of updates in the wavelet domain is $O(\log^d N)$ and $O(2^d \log N)$ for standard and non-standard Haar wavelets, respectively. A subtle issue here is that our search-tree structure operates over a linear ordering of the N^d coefficients, so we require a fast method for linearizing the multidimensional coefficient array — any simple linearization technique will work (e.g., row-major ordering or other space-filling curves).

Recall that our goal is to (approximately) recover the B most significant Haar DWT coefficients, without exhaustively searching through all coefficients. As shown in Theorem 3.8.1, creating GCSs for dyadic ranges over the (linearized) wavelet-coefficient domain, allows us to efficiently identify high-energy coefficients. (For simplicity, we fix the degree of our search structure to $r = 2$ in what follows.) An important technicality here is to select the right threshold for coefficient energy in our search process, so that our final collection of recovered coefficients provably capture most of the energy in the optimal B-term representation. Our analysis in the following theorem shows how to set this threshold, an proves that, for data vectors satisfying the "small-B property", our GCS techniques can efficiently track near-optimal approximate wavelet representations. (We present the result for the standard form of the multidimensional Haar DWT — the one-dimensional case follows as the special case $d = 1$.)

Theorem 3.8.2. *If a d-dimensional data stream over the $[N]^d$ domain has a B-term standard wavelet representation with energy at least $\eta\|a\|_2^2$, where $\|a\|_2^2$ is the entire energy, then our GCS algorithms can estimate an at-most-B-term standard wavelet representation with energy at least $(1-\epsilon)\eta\|a\|_2^2$ using space of $O(\frac{B^3 d \log N}{\epsilon^3\eta^3} \cdot \log \frac{Bd\log N}{\epsilon\eta\delta})$, per item processing time of $O(d \log^{d+1} N \cdot \log \frac{Bd\log N}{\epsilon\eta\delta})$, and query time of $O(\frac{B^3 d}{\epsilon^3\eta^3} \cdot \log N \cdot \log \frac{Bd\log N}{\epsilon\eta\delta})$.*

Proof. Use our GCS search algorithm and Theorem 3.8.1 to find all coefficients with energy at least $\frac{\epsilon\eta}{B}\|a\|_2^2 = \frac{\epsilon\eta}{B}\|\widehat{a}\|_2^2$. (Note that $\|a\|_2^2$ can be easily estimated to within small relative error from our GCSs.) Among those choose the highest B coefficients; note that there could be less than B found. For those coefficients selected, observe we incur two types of error. Suppose we choose a coefficient which is included in the best B-term representation, then we could be inaccurate by at most $\frac{\epsilon\eta}{B}\|a\|_2^2$. Now, suppose we choose coefficient c_1 which is not in the best B-term representation. There has to be a coefficient c_2 which is in the best B-term representation, but was rejected in favor of c_1. For this rejection to have taken place their energy must differ by at most $2\frac{\epsilon\eta}{B}\|a\|_2^2$ by our bounds on the accuracy of estimation for groups of size 1. Finally, note that for any coefficient not chosen (for the case when we pick fewer than B coefficients) its true energy must be less than $2\frac{\epsilon\eta}{B}\|a\|_2^2$. It follows that the total energy we obtain is at most $2\epsilon\eta\|a\|_2^2$ less than that of the best B-term representation. Setting parameters λ, ϵ', N' of Theorem 3.8.1 to $\lambda = \epsilon' = \frac{\epsilon\eta}{B}$ and $N' = N^d$ we obtain the stated space and query time bounds. For the per-item update time, recall that a single update in the original data domain requires $O(\log^d N)$ coefficient updates. □

The corresponding result for the non-standard Haar DWT follows along the same lines. The only difference with Theorem 3.8.2 comes in the per-update processing time which, in the non-standard case, is $O(d2^d \log N \cdot \log \frac{Bd\log N}{\epsilon\eta\delta})$.

3.9 Experiments

In this section we present an experimental evaluation of our techniques for conventional wavelet synopses for two data stream models. In particular, Section 3.9.1 studies the effect of the SHIFT-SPLIT operations for the time series model, whereas Section 3.9.2 investigates the performance of the group-count sketch for the update model.

3.9.1 Time Series Streams

We study the performance of the SHIFT-SPLIT operations. First, we study the effect of various parameters on the operations. Next, we show how SHIFT-SPLIT operations are employed for the maintenance of transformed data in an appending scenario. Finally, we show the significant improvement in the update cost for maintaining a wavelet synopsis for time series data streams.

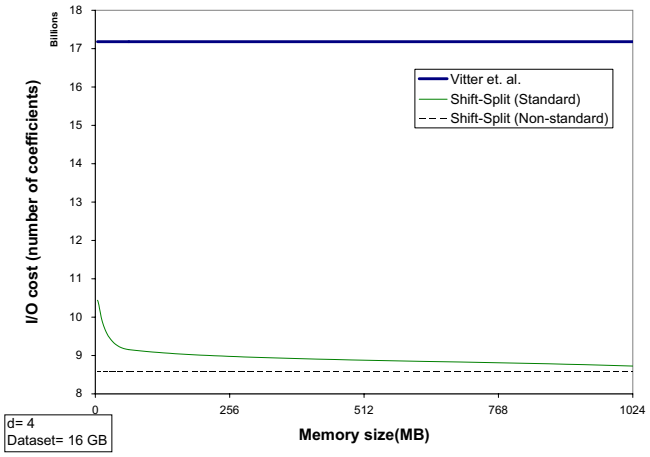


Figure 3.9: *Effect of larger memory*

Transformation of Massive Multidimensional Datasets. In this set of experiments, we transform a large dataset, **TEMPERATURE**, into the wavelet domain using limited available memory. The **TEMPERATURE** dataset is a real-world dataset provided to us by JPL that measures the temperatures at points all over the globe at different altitudes for 18 months, sampled twice every day. We construct a 4-dimensional cube with latitude, longitude, altitude and time as dimension attributes, and temperature as the measure attribute, with the total size of the cube being 16GB.

Figure 3.9 shows that larger memory considerably reduces transformation cost of SHIFT-SPLIT in the Standard form but it does not noticeably affect SHIFT-SPLIT in the Non-Standard form. The reason behind this is that the cost of the SPLIT operation is considerably different for the two forms of multidimensional wavelet transformation. Increasing memory size causes a significant decrease in SPLIT cost and consequently a major decrease of the Standard form transformation as there are many coefficients affected by the contributions of the SPLIT operation. However, SPLIT cost is almost negligible in Non-Standard form (see Table 3.1). Finally, this figure also states that our SHIFT-SPLIT approach outperforms the Vitter et al. [88] algorithm for any memory size.

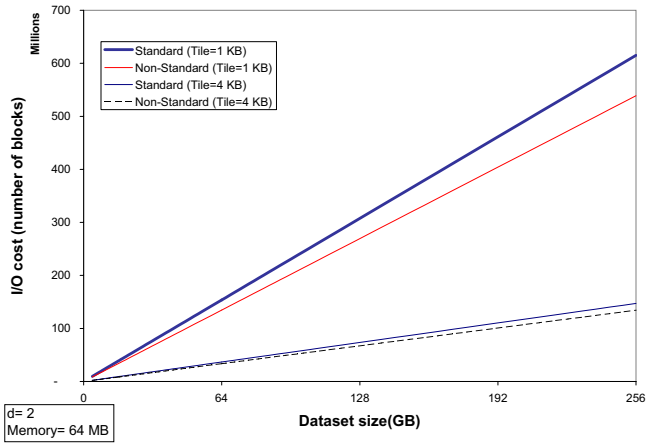


Figure 3.10: *Effect of larger tiles*

As we have shown in Section 3.4, not only Tiling is the optimal wavelet coefficient blocking for query processing, but it is also a SHIFT-SPLIT friendly schema which introduces significant cost improvements in the transformation process. Figure 3.10 demonstrates this fact, by using different tile sizes and thus illustrates the scalability of SHIFT-SPLIT algorithm.

Appending to Wavelet-Transformed Data. We examine our proposed appending technique on the **PRECIPITATION** [91] dataset, where we incrementally receive new sets of data every month. **PRECIPITATION** is a real-life dataset that measures the daily precipitation for the Pacific Northwest for 45 years. We built a 3-dimensional cube with latitude, longitude and time as dimensional attributes, and precipitation as the measure attribute for every day. The sizes of these dimensions are 8,8 and 32 respectively for each month. Figure 3.11 demonstrates the SHIFT-SPLIT I/O cost as new sets of data are appended. The sudden jumps in the figure correspond to the expansion process, where all coefficients must be shifted to accommodate for new data values. One can observe that this expansion process is not such a dominating factor as described in Section 3.5.2, especially for larger disk block sizes.

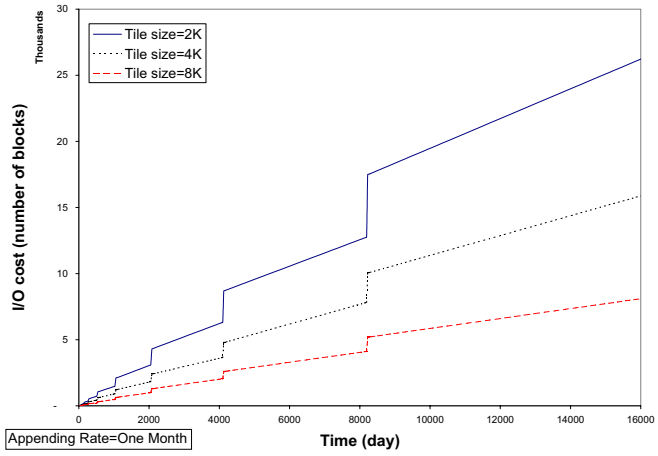


Figure 3.11: *Appending in synopses*

Data Stream Synopses. In this scenario we only need to preserve the synopsis of the **PRECIPITATION** dataset, limited to a memory footprint of 40KB. Figure 3.12 demonstrates the computational cost versus the extra storage trade-off described in Section 3.3. As the figure suggests, the update cost can be improved by 88% by employing additional buffer memory of only 6% of the total synopsis size.

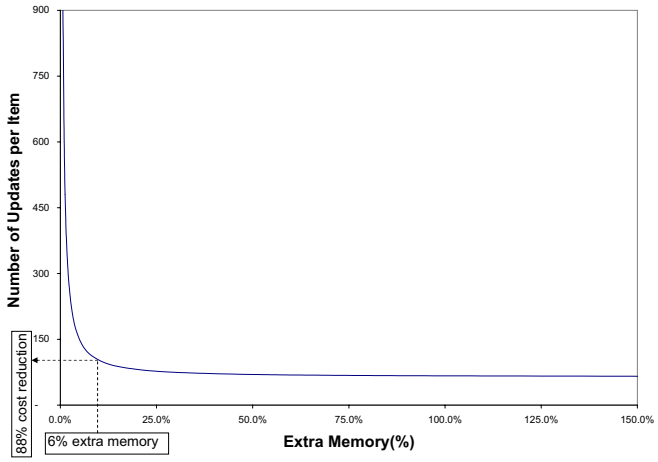


Figure 3.12: *SHIFT-SPLIT in multidimensional streaming*

3.9.2 Update Streams

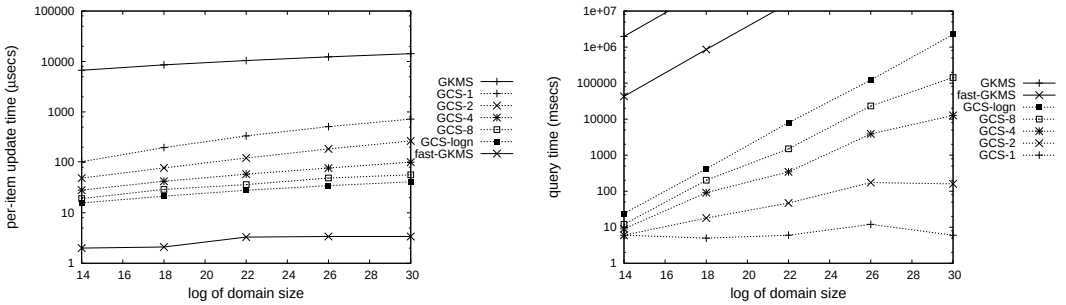
Datasets and Methodology. We implemented our algorithms in a mixture of C and C++, for the Group-Count sketch (GCS) with variable degree. For comparison we also implemented the method of [40] (GKMS) as well as a modified version of the algorithm with faster update performance using ideas similar to those in the Group-Count sketch, which we denote by fast-GKMS. Experiments were performed on a 2GHz processor machine, with 1GB of memory. We worked with a mixture of real and synthetic data:

- *Synthetic Zipfian Data* was used to generate data from arbitrary domain sizes and with varying skewness. By default the skewness parameter of the distribution is $z = 1.1$.
- *Meteorological dataset* [3] comprised of 10^5 meteorological measurements. These were quantized and projected appropriately to generate datasets with dimensionalities between 1 and 4. For the experiments described here, we primarily made use of the `AirTemperature` and `WindSpeed` attributes to obtain 1- and 2-dimensional data streams.

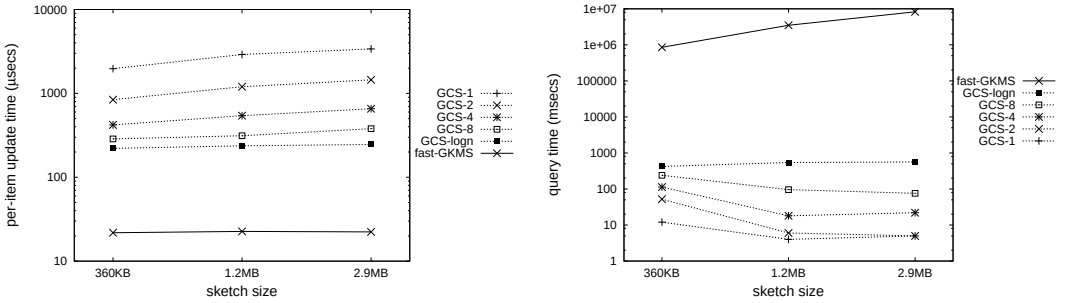
In our experiments, we varied the domain size, the size of the sketch[4] and the degree of the search tree of our GCS method and measured (1) per-item update time, (2) query time and (3) accuracy. In all figures, GCS-k denotes that the degree of the search tree is 2^k; i.e. GCS-1 uses a binary search tree, whereas GCS-logn uses an n-degree tree, and so has a single level consisting of the entire wavelet domain.

[3] `http://www-k12.atmos.washington.edu/k12/grayskies/`

[4] In each experiment, all methods are given the same total space to use.

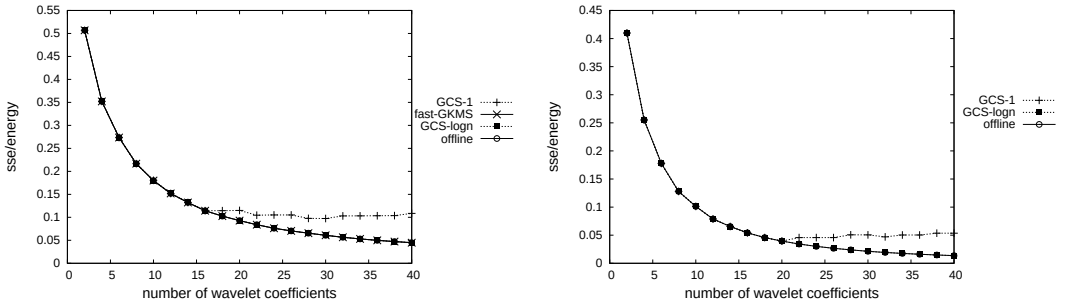


(a) Update Time against domain size (b) Query Time against domain size



(c) Update Time against space (d) Query Time against space

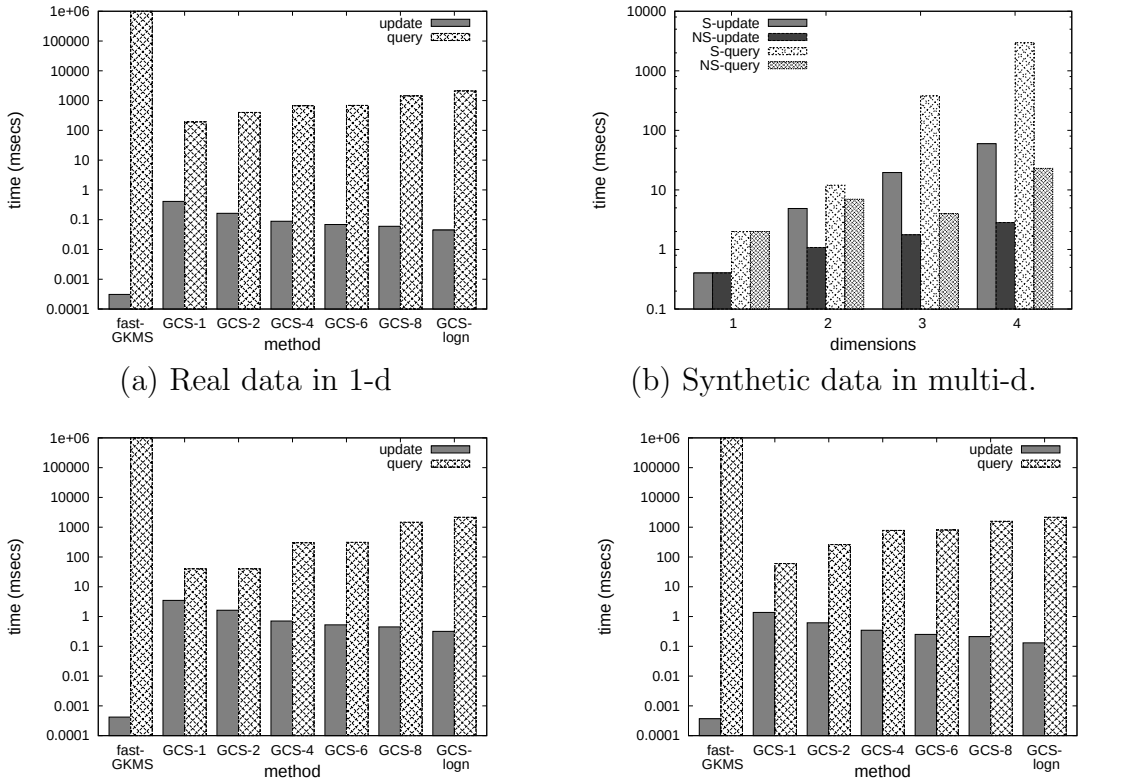
Figure 3.13: *Performance on one-dimensional data*



(a) z=1.1 (b) z=1.4

Figure 3.14: *Accuracy of wavelet synopses*

One-Dimensional Experiments. In the first experimental setup we used a synthetic 1-dimensional data stream with updates following the Zipfian distribution ($z = 1.1$). Space was increased based on the log of the dimension, so for $\log N = 14$, 280KB was used, up to 600KB for $\log N = 30$. Figure 3.13 (a) shows the per-item update time for various domain sizes, and Figure 3.13 (b) shows the time required to perform a query, asking for the top-5 coefficients. The GKMS method takes orders of magnitude longer for both updates and queries, and this behavior is seen in all other experiments, so we do not consider it further. Apart from this, the ordering (fastest to slowest) is reversed between update time and query time. Varying the degree of the search tree allows update time and query time to be traded off. While the fast-GKMS approach is the fastest for updates, it is dramatically more expensive for queries, by several orders of magnitude. For domains of size 2^{22}, it takes several hours to recover the coefficients, and extrapolating to a 32 bit domain means recovery would take over a week. Clearly this is not practical for realistic monitoring scenarios. Although GCS-logn also performs exhaustive search over the domain size, its query times are significantly lower as it does not require a sketch



(a) Real data in 1-d

(b) Synthetic data in multi-d.

(c) Real data in 2-d (standard DWT)

(d) Real data in 2-d (Non-standard DWT).

Figure 3.15: *Performance on 1-d real and multi-d real and synthetic data*

construction and inner-product query per wavelet coefficient.

Figures 3.13 (c) and (d) show the performance as the sketch size is increased. The domain size was fixed to 2^{18} so that the fast-GKMS method would complete a query in reasonable time. Update times do not vary significantly with increasing space, in line with our analysis (some increase in cost may be seen due to cache effects). We also tested the accuracy of the approximate wavelet synopsis for each method. We measured the SSE-to-energy ratio of the estimated B-term synopses for varying B and varying Zipfian parameter and compared it against the optimal B-term synopsis computed offline. The results are shown in Figures 3.14 (a) and (b), where each sketch was given space 360KB. In accordance to analysis (GCS requires $O(\frac{1}{\epsilon})$ times more space to provide the same guarantees with GKMS) the GCS method is slightly less accurate when estimating more than the top-15 coefficients. However, experiments showed that increasing the size to 1.2MB resulted in equal accuracy. Finally we tested the performance of our methods on single dimensional meteorological data of domain size 2^{20}. Per-item and query times in Figure 3.15 (a) are similar to those on synthetic data.

Multidimensional Experiments. We compared the methods for both wavelet decomposition types in multiple dimensions. First we tested our GCS method for a synthetic dataset ($z = 1.1$, 10^5 tuples) of varying dimensionality. In Figure 3.15 (b) we kept the total domain size constant at 2^{24} while varying the dimensions between 1 and 4. The per-item update time is higher for the standard decomposition, as there are more updates on the wavelet domain per update on the original domain. The increase in query time can be attributed to the increasing sparseness of the domain as the dimensionality increases which makes searching for big coefficients

harder. This is a well known effect of multidimensional standard and non-standard decompositions. For the real dataset, we focus on the two dimensional case; higher dimensions are similar. Figure 3.15(c) and (d) show results for the standard and non-standard respectively. The difference between GCS methods and fast-GKMS is more pronounced, because of the additional work in producing multidimensional wavelet coefficients, but the query times remain significantly less (query times were in the order of hours for fast-GKMS), and the difference becomes many times greater as the size of the data domain increases.

The Group-Count sketch approach is the only method that achieves reasonable query times to return an approximate wavelet representation of data drawn from a moderately large domain (2^{20} or larger). Our first implementation is capable of processing tens to hundreds of thousands of updates per second, and giving the answer to queries in the order of a few seconds. Varying the degree of the search tree allows a tradeoff between query time and update time to be established. The observed accuracy is almost indistinguishable from the exact solution, and the methods extend smoothly to multiple dimensions with little degradation of performance.

3.10 Summary

This chapter introduced techniques for constructing conventional wavelet synopses for two data stream models. In particular, we have introduced two general purpose operations, termed SHIFT and SPLIT, that work directly in the wavelet domain and apply for time-series data streams. We analyze costs for both the single dimensional case and the two forms of multidimensional transformation. There is a significant number of applications that can benefit from these operations. We have revisited some data maintenance scenarios, such as transforming massive multidimensional datasets and reconstructing large ranges from wavelet decomposed data, and utilized the SHIFT-SPLIT operations to draw comparisons with current state of the art techniques. Furthermore, we have provided solutions to some previously un-explored maintenance scenarios, namely, appending data to an existing transformation and approximation of multidimensional data streams. We demonstrated the effectiveness of the proposed techniques both analytically and experimentally, and we conjecture that the introduced operations can prove useful in a plethora of other applications, as the SHIFT-SPLIT operations stem from the general properties and behavior of wavelets.

Regarding the update stream model, we have proposed the first known streaming algorithms for space- and time-efficient tracking of approximate wavelet summaries for both single and multidimensional data. Our approach relies on a novel, Group-Count Sketch (GCS) synopsis that, unlike earlier work, satisfies all three key requirements of effective streaming algorithms, namely: (1) polylogarithmic space usage, (2) small, logarithmic update times (essentially touching only a small fraction of the GCS for each streaming update); and, (3) polylogarithmic query times for computing the top wavelet coefficients from the GCS. Our experimental results with both synthetic and real-life data have verified the effectiveness of our approach, demonstrating the ability of GCSs to support very high speed data sources.

Chapter 4

Hierarchically Compressed Wavelet Synopses

In this chapter we present an alternative method for storing wavelet synopses. Unlike conventional synopses discussed in Chapter 3, the concept of *Hierarchically Compressed Wavelet Synopses* (HCWS) reduces the memory footprint for storing and indexing wavelet coefficients. Therefore, under the same space restrictions, HCWS manages to capture more information about the summarized data leading to smaller reconstruction errors (SSE).

Existing research on stream summarization studies focus on selecting an optimal set of wavelet coefficients to store so as to minimize some error metric, without however seeking to reduce the size of the wavelet coefficients themselves. In many real datasets the existence of large spikes in the data values results in many large coefficient values lying on paths of a conceptual tree structure known as the wavelet tree. To exploit this fact, we introduce a novel compression scheme for wavelet synopses, termed Hierarchically Compressed Wavelet Synopses, that fully exploits hierarchical relationships among coefficients in order to reduce their storage. Our proposed compression scheme allows for a larger number of coefficients to be stored for a given space constraint thus resulting in increased accuracy of the produced synopsis. We propose optimal, approximate and greedy algorithms for constructing hierarchically compressed wavelet synopses that minimize SSE while not exceeding a given space budget for time series data streams. Extensive experimental results on both synthetic and real-world datasets validate our novel compression scheme and demonstrate the effectiveness of our algorithms against existing synopsis construction algorithms.

The remainder of this chapter is organized as follows. Section 4.1 contains the motivation for this chapter and reviews related work. Section 4.2 builds the necessary background on wavelet decomposition, introduces the concept of Hierarchically Compressed Wavelet Synopses and formally presents our optimization problem. In Section 4.3 we formulate a dynamic programming recurrence and use it to optimally solve this optimization problem. Next, in Section 4.4 we present an approximation algorithm with tunable guarantees, whereas, in Section 4.5 we present a faster greedy algorithm. In Section 4.6 we provide a streaming version of our greedy algorithm. In Section 4.7 we sketch some useful extensions of our algorithms and in Section 4.8 we describe the results of our empirical study. Finally, Section 4.9 provides some concluding remarks and future directions.

4.1 Motivation and Related Work

In conventional wavelet synopses, the selected coefficients are stored as pairs $\langle Coords, Value\rangle$, where the first element ($Coords$) is the *coordinates/index* of the coefficient and determines the data that this coefficient helps reconstruct (also termed as the *support region* of the coefficient), while the second element ($Value$) denotes the magnitude/value of the coefficient. Depending on the actual storage representation for these elements (i.e., integer values for the coordinates and floating point numbers for the coefficient value) and the data dimensionality, the fraction of the available storage for the synopsis that is used for storing coefficient coordinates can be significant. If sizeof($Coord$) and sizeof($Value$) denote the storage requirements for the coefficient coordinates[1] and the coefficient value, correspondingly, then the storage of the coefficient coordinates will occupy a fraction $\frac{\text{sizeof}(Coord)}{\text{sizeof}(Coord)+\text{sizeof}(Value)}$ of the overall synopsis size (see Section 4.2).

While reducing the storage overhead of the wavelet coordinates would allow for a larger number of coefficient values to be stored, and would thus result in increased accuracy of the synopsis, to our knowledge none of the above techniques tries to exploit this fact and incorporate it in the coefficient thresholding process. A past suggestion [8] has proposed selecting for storage only the top coefficient values (i.e., the ones with the largest support regions). Using such an approach, no coordinates need to be stored. However, such an approach does not give any guarantee on whether the selected coefficients can significantly reduce the desired error metric. Finally, techniques that target, possibly multidimensional, datasets with multiple measures [23, 44] exploit storage dependencies among only coefficient values that correspond to the same coordinates, but for different measures.

To address the drawbacks of existing techniques, we propose a novel, flexible, compression scheme, termed *Hierarchically Compressed Wavelet Synopses* (denoted as HCWS), for storing wavelet coefficients. In a nutshell, instead of individually storing wavelet coefficients, our compression scheme allows for storing sets of coefficient values. These stored sets are not arbitrary, but are rather composed by coefficients that lie on a path of the wavelet tree. Each path of coefficient values stored as a hierarchically compressed wavelet coefficient (HCC) can be uniquely identified by (i) the coordinates of the path's lowest, in the wavelet tree, stored coefficient LC; and (ii) a bitmap that reveals how many ancestors of LC are also stored in the same HCC. Utilizing such an index-sharing setting leverages better space allocation, since the coordinates of a single coefficient need to be stored in each path, which can result to increased accuracy of the obtained approximation. To briefly illustrate the benefits of our approach, consider the sample wavelet tree depicted in Figure 4.1. In this figure, the values of 16 coefficients are depicted, using the symbol c_i to denote the coefficient at coordinate i. Assuming a space budget of 41 bytes, and using 8 bytes for storing the$\langle Coord, Value\rangle$ pairs, the optimal conventional wavelet synopsis would simply store the coefficients c_0, c_1, c_7, c_8 and c_{15} shown in gray. On the other hand, our hierarchically compressed wavelet synopsis, given the same space budget, would store the two paths shown in Figure 4.1 — that is, it would manage

[1] While for a D-dimensional dataset, the D coefficient coordinates could be stored uncompressed, alternative encodings can be utilized to limit their size. For example, utilizing a location function for arrays, the D-dimensional coefficient coordinates can be encoded with space that depends on the product of the dimension cardinalities.

to also store coefficients c_2, c_3, c_5 and c_{11} in comparison to the coefficient c_8 selected by the conventional wavelet synopsis. The effect of including these coefficients is the reduction of the sum squared error (SSE) of the approximation by 60% (SSE of 294 instead of 752).

A question that naturally arises is whether "important", for the desired error metric, coefficient values can frequently occur within such a path and would, thus, be beneficial to store using a HCC. As we explain in Section 4.2, due to the nature of the wavelet decomposition process, this behavior is expected to be frequently observed, and only, in datasets with frequent spikes and discontinuities in neighboring domain regions. These discontinuities are often due to large spikes in the collected data values, such as the ones observed in network monitoring applications where the number of network packets may often exhibit a bursting behavior. A similar behavior also occurs in sparse regions over large domain sizes, where either few non-zero data values may occur in an otherwise empty region, or where dense regions neighbor empty regions of the data.

A closely related concept to hierarchically compressed synopses is the notion of extended wavelets [22, 23] for the case of datasets with multiple measures; some further improvements were presented in [44], where a streaming algorithm for the above problem is also introduced. A common characteristic of the work in [22, 23, 44] with this chapter is that all of these papers seek to exploit storage dependencies amongst stored coefficient values. However, these storage dependencies are only amongst coefficient values, of different measures, that correspond to the same coefficient coordinates. Thus, the storage overhead of a coefficient value is not influenced by whether other coefficient values in the path towards the root of the wavelet tree have also been stored. This observation implied that the wavelet tree structure does not need to be taken into account at all. Due to this crucial difference with the problem tackled in this chapter, the techniques in [22, 23, 44] cannot be used to solve our optimization problem, and are in fact completely different than the techniques that we propose here. Similarly, extending our proposed algorithms of this chapter to multi-measure datasets requires significant modifications and is an interesting topic of future work. I/O efficient algorithms for maintenance tasks were presented in [50].

In a previous work [8], the authors proposed the storage of coefficient values forming a rooted subtree of the wavelet tree. While such an approach was guaranteed to provide a worse benefit than the conventional thresholding process, their techniques performed well for signal de-noising purposes. However, this work neither considered reducing the storage overhead of the wavelet coefficients' coordinates, nor did it incorporate such an objective in the thresholding process. Moreover, the requirement that rooted subtrees be stored, rather than arbitrary paths of coefficient values, often required the storage of many small coefficient values that simply happened to lie on root-to-leaf paths of other large coefficient values.

A lot of recent work focus on constructing wavelet synopses that minimize error metrics other than SSE. The work in [32] constructs wavelet synopses that probabilistically minimize the maximum relative or absolute error incurred for reconstructing any data value. The work in [21] provides a sparse approximation scheme for the same problem. While solving entirely an entirely different problem, our HCApprL2 algorithm shares in fact several common characteristics in its operation with the algorithm in [21]. However, the HCApprL2 algorithm is slightly more com-

plicated due to the two mutually recursive functions that it needs to approximate, and the increased number of breakpoint combinations of children nodes that it needs to consider in its operation. Such details also lead to a more tedious proof of its correctness.

The contributions of our work can be summarized as follows.

1. We introduce the concept of HCWS, a novel compression scheme that fully exploits the hierarchical relationships among wavelet coefficients, and that may lead to significant accuracy gains.

2. We propose a novel, optimal dynamic programming algorithm, HCDynL2, for selecting the HCWS that minimizes the sum of squared errors under a given synopsis size budget. We then propose a streaming variant of the optimal algorithm that can operate in one pass over the data using limited memory.

3. We present an approximation algorithm, HCApprL2, with tunable guarantees, for the *benefit* of the obtained solution, for the same optimization problem. Further, we present a streaming variant of the algorithm.

4. Due to the large running time and space requirements of our DP solution, we introduce a fast greedy, HCGreedyL2, algorithm with space and time requirements on par with conventional synopsis techniques. We then also present a streaming variant, the HCGreedyL2-Str algorithm, of the greedy algorithm.

5. We sketch useful extensions for multidimensional datasets and running time improvements for large domain sizes.

6. We present extensive experimental results of our algorithms on both synthetic and real-life datasets. Our experimental study demonstrates that (i) the use of HCWS can lead to wavelet synopses with significantly reduced errors; (ii) HCApprL2 constructs HCWS with tunable accuracy guarantees; (iii) although HCGreedyL2 cannot provide guarantees in the quality of the obtained synopsis, it always provides near-optimal solutions, while exhibiting very fast running times; and (iv) The HCGreedyL2-Str algorithm consistently provides results comparable to those of the HCGreedyL2 algorithm.

4.2 Hierarchical Compression

In this section, we quickly discuss the existing strategies for obtaining wavelet synopses and demonstrate some of their important shortcomings. Then, we introduce the notion of *hierarchically compressed wavelet coefficients and synopses*, which form the basis for our proposed approach and data-reduction algorithms. Finally, we formally define the problem we address in the remainder of this chapter.

We refer to the following example to draw comparisons between conventional and hierarchically compressed wavelet synopses. Suppose we are given the one-dimensional data vector A containing the $N = 16$ data values $A = [17, 41, 32, 30, 36, 36, 35, 57, 0, 0, 0, 0, 0, 0, 0, 36]$. The one-dimensional Haar wavelet transform of A is given by $W_A = [20, 15.5, -5.5, -4.5, -1, -5, 0, -9, -12, 1, 0, -11, 0, 0, 0, -18]$. Figure 4.1 depicts the wavelet tree for our example data vector A.

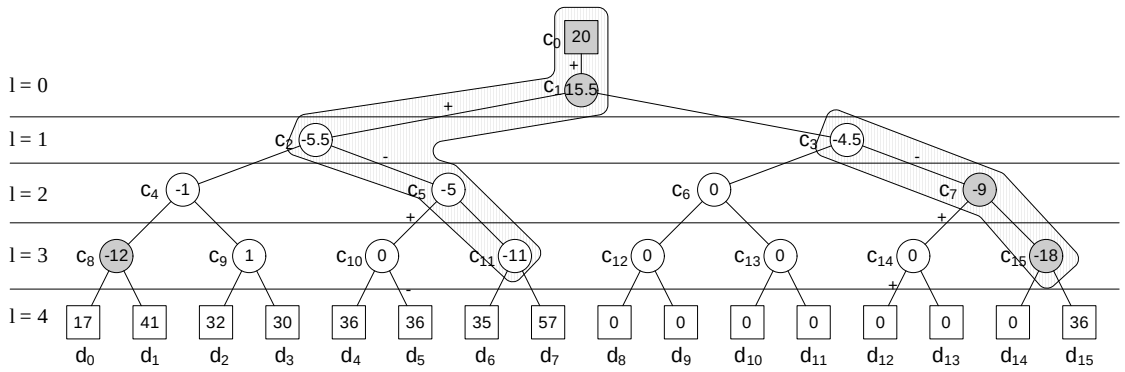


Figure 4.1: *Wavelet tree structure for example data vector A*

Table 4.1 summarizes some of the key notational conventions used throughout this chapter; additional notation is introduced when necessary. Detailed symbol definitions are provided at the appropriate locations in the text.

Symbol	**Description** $(i \in \{0, \ldots, N-1\})$
N	Number of data-array cells
D	Data-array dimensionality
B	Space budget for synopsis
A, W_A	Input data and wavelet transform arrays
d_i	Data value for i^{th} cell of data array
$\hat{d}_i$	Reconstructed data value for i^{th} cell
c_i, c_i^*	Un-normalized/normalized Haar coefficient coordinate i
path(u)	Set of non-zero proper ancestors of u in the wavelet tree
level(c_i)	The level of the wavelet tree c_i belongs to
HCC	A hierarchically compressed wavelet coefficient
bottom(HCC)	The bottommost coefficient that belongs to HCC
top(HCC)	The topmost coefficient that belongs to HCC
parent(HCC)	The parent of the topmost coefficient that belongs to HCC

Table 4.1: *Notation*

Given a node u in an wavelet tree T, let path(u) denote the set of all proper ancestors of u in T (i.e., the nodes on the path from u to the root of T, including the root but not u) with non-zero coefficients. A key property of the Haar wavelet decomposition is that the reconstruction of any data value d_i depends only on the values of coefficients on path(d_i). For example, in Figure 4.1, $d_5 = c_0 + c_1 - c_2 + c_5 = 20 + 15.5 - (-5.5) + (-5) = 36$. Note that, intuitively, wavelet coefficients carry different weights with respect to their importance in rebuilding the original data values. For example, the overall average and its corresponding detail coefficient are obviously more important than any other coefficient since they affect the reconstruction of all entries in the data array. In order to weigh the importance of all wavelet coefficients, we need to appropriately *normalize* the final entries of W_A. A common normalization scheme [85] is to multiply each wavelet coefficient c_i by $\sqrt{2^{\log N - \text{level}(c_i)}}$, where level($c_i$) denotes the *level of resolution* at which the coefficient appears (with 0 corresponding to the "coarsest" resolution level and $\log N$ to the "finest"). Given this normalization procedure, the normalized values of the wavelet coefficients of our example data array A are: $[80,\ 62,\ -11\sqrt{2},\ -9\sqrt{2},\ -2,\ -10,\ 0,\ -18,\ -12\sqrt{2},\ \sqrt{2},\ 0,\ -11\sqrt{2},\ 0,\ 0,\ 0,\ -18\sqrt{2}]$.

Given a limited amount of storage for building a *wavelet synopsis* of the input data array A, the conventional thresholding procedure retains a certain number $B_C \ll N$ of the coefficients in W_A as a highly-compressed approximate representa-

tion of the original data (the remaining coefficients are implicitly set to 0). The goal of coefficient thresholding is to determine the "best" subset of B_C coefficients to retain, so that some overall error measure in the approximation is minimized. The method of choice for the vast majority of studies on wavelet-based data reduction and approximation [13, 64, 65] is *conventional coefficient thresholding* that greedily retains the B_C largest Haar-wavelet coefficients in *absolute normalized value*. This thresholding method *provably* minimizes the sum squared error (SSE). Indeed, in a mathematical view point, the process of computing the wavelet transform and normalizing the coefficients is actually the orthonormal transformation of the data vector with respect to the Haar basis. Parseval's formula guarantees that choosing the B_C largest coefficients is optimal with respect to the SSE. Consider our example array A and assume that we have a space budget of 41 bytes. In conventional synopses we require to store each coefficient as a $\langle i, c_i \rangle$ pair, where i denotes the index/coordinate of the coefficient and c_i denotes its value. Thus, our budget translates to 5 coefficients, if we further assume that a coordinate and a coefficient value cost 4 bytes each. Optimizing for the sum of squared errors, leads to choosing the 5 largest (in absolute normalized value) coefficients. These retained coefficients c_0, c_1, c_7, c_8 and c_{15} are shown in gray in Figure 4.1. Note that in D-dimensional datasets the stored coefficients consist of the D dimension coordinates (which, as mentioned in Section 4.1, can be stored in less space than explicitly storing them as D integer values) and of the coefficient"s value.

As discussed in Section 4.1, the main drawback of conventional wavelet synopses for minimizing the SSE of the approximation is that not only is there no effort to reduce the storage overhead of the selected coefficients but, more importantly, that this objective is not incorporated in the operation of the algorithm. The same drawback also occurs in thresholding algorithms that try to minimize other error metrics, such as the maximum or weighted sum squared absolute/relative error of the approximation [32, 33, 41, 73, 62]. Due to the differencing process employed by the wavelet decomposition between average values of neighboring regions, multiple large coefficient values may exhibit hierarchical relationships (i.e., belong in the same path) only when spikes over some regions of the data are large enough[2] to significantly impact the values of coefficients (and, thus, generate coefficients with large values) in multiple (and potentially all) resolution levels. Datasets which include multiple spikes with the aforementioned property (i.e., can generate multiple large coefficients in their path), present great opportunity for exploiting the hierarchical relationships among important coefficient values and also provide better opportunities for our presented techniques to be most effective.

Given the shortcomings of the existing wavelet thresholding algorithms we now introduce the notion of a *hierarchically compressed wavelet coefficient* (HCC). For ease of presentation, we initially focus on the one-dimensional case. The extensions to multidimensional datasets are presented in Section 4.7.

Definition 4.2.1. *A hierarchically compressed (HCC) wavelet coefficient is a triplet* $\langle \text{Bit}, C, V \rangle$ *consisting of:*

- *A bitmap* Bit *of size* $|\text{Bit}| \geq 1$*, denoting the storage of exactly* $|\text{Bit}|$ *coefficient values.*

[2]Besides its magnitude, the impact of a spike may also depend, in the case of the $\mathcal{L}_2^w$ error metric discussed in Section 4.7.3, on the weight specified for each data point.

- *The coordinate/index C of the bottommost stored coefficient.*
- *The set V of $|\text{BIT}|$ stored coefficient values.*

The bitmap of a HCC can help determine how many coefficient values have actually been stored. By representing the number of stored coefficients in unary format, as a series of $(|V| - 1)$ 1-bits and utilizing a 0-bit as the last bit (also acting as a stop bit), any hierarchically compressed wavelet coefficient that stores $|V|$ coefficient values requires a bitmap of just $|V|$ bits. A *hierarchically compressed wavelet synopsis* (HCWS) consists of a set of HCCs, in analogy to a conventional synopsis that comprises $\langle Coords, Value \rangle$ pairs.

Returning to our example array A, for a space budget of 41 bytes, or 328 bits, optimizing for the SSE metric results in storing two hierarchically compressed coefficients. These HCCs are essentially the two paths illustrated in Figure 4.1 and are depicted in Table 4.2. Assuming, as before, that a coordinate and a coefficient value each require 32 bits, the first hierarchical coefficient requires $32 + 5 + 5 \cdot 32 = 197$ bits, whereas the second one requires $32 + 3 + 3 \cdot 32 = 131$ bits.

Coordinate	Bitmap	Set of Coefficient Values
11	11110	{−11, −5, −5.5, 15.5, 20}
15	110	{−18, −9, −4.5}

Table 4.2: *HCWS for data vector A and $B = 41$ bytes*

It is easy to see how a hierarchically compressed synopsis better utilizes the available space, and in doing so manages to store 3 more coefficients than the conventional synopsis retains. In terms of SSE, the conventional synopsis loses 752, whereas the HCWS just 294 — an improvement of over 60%. It is important though to emphasize that the coefficient values stored in HCWS are not necessarily a superset of the coefficients selected by the conventional thresholding algorithm, since it is often more beneficial to exploit storage dependencies and store multiple coefficient values that lie on a common path, than storing a slightly larger individual value, as shown in Section 4.8. In our example, note that the c_8 coefficient, selected by a conventional synopsis, is not included in the optimal HCWS.

The selection of which hierarchically compressed wavelet coefficients to store is based on the optimization problem we are trying to solve. To simplify notation, in our discussion hereafter the unit of space is set equal to 1 bit, and all space requirements are expressed in terms of this unit. The bulk of the work in wavelet-based compression of data tries to minimize the sum of squared absolute errors (SSE) of the overall approximation. We focus on the same problem, here; extensions to the sum of squared relative errors, or any weighted $\mathcal{L}_2^w$ norm, can be found in Section 4.7. More formally, the optimization problem can be posed as follows:

Problem 4.1 [Sum of Squared Errors Minimization for Hierarchically Compressed Coefficients] Given a collection W_A of wavelet coefficients and a storage constraint B select a synopsis $\mathcal{S}$ of hierarchically compressed wavelet coefficients HCC's that minimizes the sum of squared errors; that is, minimize $\sum_{i=0}^{N-1} (d_i - \hat{d}_i)^2$ subject to the constraint $\sum_{HCC \in \mathcal{S}} |HCC| \leq B$, where $|HCC|$ denotes the space requirement for storing an HCC. □

Based on Parseval's theorem and the previous discussion, using c_i^* to denote the

normalized value for the i^{th} wavelet coefficient, we can restate the above optimization problem in the following equivalent (but easier to process) form.

Problem 4.2 [Benefit Maximization for Hierarchically Compressed Coefficients] Given a collection W_A of wavelet coefficients and a storage constraint B, select a synopsis $\mathcal{S}$ of hierarchically compressed wavelet coefficients that maximizes the sum $\sum_{i=0}^{N-1} (c_i^*)^2$ of the squared retained normalized coefficient values, subject to the constraint $\sum_{HCC \in \mathcal{S}} |HCC| \leq B$, where $|HCC|$ denotes the space requirement for storing an HCC. □

4.3 HCDynL2: An Optimal Dynamic-Programming Algorithm

We now propose a thresholding algorithm (termed HCDynL2) based on Dynamic-Programming (DP) ideas, that optimally solves the optimization problem described above. Our HCDynL2 algorithm takes as input a set of input coefficient values W_A and a space constraint B. HCDynL2 then selects an optimal set of hierarchically compressed coefficients for Problem 4.2. Before explaining the operation of our HCDynL2 algorithm, we need to introduce the notion of *overlapping* paths.

Definition 4.3.1. *Two paths are* overlapping *if they both store the value of at least one common coefficient.*

It is important to note that the benefit of storing two overlapping paths is not equal to the sum of benefits of these two paths, since the storage of at least one coefficient value is duplicated. Thus, the benefit of each path depends on which other overlapping paths are included in the optimal solution. The possibly varying benefit of each candidate path is the main difficulty in formulating an optimal algorithm. To make matters worse, the number of candidate paths that may be part of the solution is quite large ($O(N \log N)$), as is the number of overlapping paths. In particular, any coefficient value c_i belonging at level level(c_i) may be stored in up to $\sum_{\substack{0 \leq k \leq \min\{j, \text{level}(c_i)\} \\ \text{level}(c_i) - k + j \leq \log N}} 2^k$ paths of length $1 \leq j \leq \log N + 1$ (i.e., paths originating from nodes in its subtree with distance at most j from c_i). Fortunately, the following lemma helps reduce the search space of our algorithm, by considering the structure of the wavelet tree.

Lemma 4.3.1. *The optimal solution for Problem 4.2 (and equivalently for Problem 4.1) never needs to consider overlapping paths.*

The proof of Lemma 4.3.1 is simple and is based on the observation that for any solution that includes a pair of overlapping paths (the extension to having multiple overlapping paths is straightforward), there exists an alternative solution with non-overlapping paths that stores exactly the same coefficient values and, thus, has the same benefit while requiring less space. This solution is produced by simply removing from one of the overlapping paths its intersection with the other path. Let the storage overhead cost of the coefficient coordinate be assigned to the lowest coefficient of each path. Thus, the required space for this coefficient (i.e., the "start-up" cost for any HCC) is $S_1 = \text{sizeof}(Coord) + \text{sizeof}(Value) + 1$ and the corresponding space for all other coefficient values in its path is simply: $S_2 = \text{sizeof}(Value) + 1$.

Symbol	Description
S_1	sizeof$(Coords)$ + sizeof$(Value)$ + 1
S_2	sizeof$(Value)$ + 1
M$[i, B]$	The optimal benefit acquired when assigning at most B space to the subtree of coefficient c_i
F$[i, B]$	The optimal benefit acquired when assigning at most B space to the subtree of coefficient c_i and when c_i is forced to be stored.

Table 4.3: *Notation used in* HCDynL2 *Algorithm*

$$\mathrm{F}[i,B] = \begin{cases} -\infty & \text{, if } i \geq N \text{ or } B < S_1 \\ \max \left\{ \begin{array}{l} \max\limits_{0 \leq b_L \leq B-S_1} (c_i^*)^2 + \mathrm{M}[2i, b_L] + \mathrm{M}[2i+1, B-b_L-S_1] \\ \max\limits_{0 \leq b_L \leq B-S_2} (c_i^*)^2 + \mathrm{F}[2i, b_L] + \mathrm{M}[2i+1, B-b_L-S_2] \\ \max\limits_{0 \leq b_L \leq B-S_2} (c_i^*)^2 + \mathrm{M}[2i, b_L] + \mathrm{F}[2i+1, B-b_L-S_2] \end{array} \right\} & \text{, otherwise} \end{cases}$$

$$\mathrm{M}[i,B] = \begin{cases} 0 & \text{, if } i \geq N \text{ or } B < S_1 \\ \max \left\{ \begin{array}{l} \max\limits_{0 \leq b_L \leq B} \mathrm{M}[2i, b_L] + \mathrm{M}[2i+1, B-b_L] \\ \mathrm{F}[i,B] \end{array} \right\} & \text{, otherwise} \end{cases}$$

(4.1)

Then, when considering the optimal solution at any node $i \geq 1$ (The extension to node 0 that has just one subtree is straightforward) of the wavelet tree, given any space constraint B, the following cases may arise:

1. Coefficient c_i is not part of the optimal solution. The optimal solution arises from the best allotment of the space B to the two subtrees of c_i.
2. Coefficient c_i is part of the optimal solution but is not a part of any hierarchically compressed path originating from any of its descendants in the wavelet tree. The optimal solution arises from storing c_i in a new hierarchically compressed path and considering the best allotment of the space $B - S_1$ to the two subtrees of c_i.
3. Coefficient c_i is part of the optimal solution and is part of a single hierarchically compressed path originating from one of its descendants that may reside in its left (right) subtree. The optimal solution arises from attaching c_i to the hierarchically compressed path of the left (right) subtree and considering the best allotment of the space $B - S_2$ to the two subtrees of c_i. However, for this space distribution process to be valid, we need to make sure that the solution that is produced by allocating space $0 \leq b \leq B - S_2$ to the left (right) subtree stores the coefficient c_{2i} (c_{2i+1}) — otherwise, c_i cannot be attached to a path originating from that subtree.

Cases 1 and 2 are pretty straightforward, since they introduce a recursive procedure that can be used to calculate the optimal solution at node i. This recursive procedure will check all possible allocations of space to the two subtrees of i and

calculate the optimal solutions in these subtrees, given the space allocated to them. The optimal solution arises from the space allocation that results in the largest benefit. Note that in these two cases there are no dependencies or requirements from the solutions sought in the two subtrees, other that they result in the largest possible benefit, given the space allocated to them (and thus seeking the optimal solutions in these subtrees suffices).

On the contrary, in Case 3 coefficient c_i, for any space allocation to its two subtrees, needs to be attached to a solution that is produced at one of its subtree **and** where this solution stores the coefficient value at the root of the subtree. Given this requirement, the solution for this subtree is not necessarily the optimal one, but only the optimal solution, given that the root of the subtree is stored. This implies that our algorithm will need to also keep track of some suboptimal solutions, similarly to the dynamic programming algorithm in [23], which seeks to exploit storage dependencies in datasets with multiple measures *only* among coefficient values of different measures that share the same coefficient coordinates (and, thus, cannot be used for the problem addressed in this chapter). On the other hand, the goal of our algorithm is to explore hierarchical relationships among coefficient values of different coordinates in order to reduce their storage overhead and improve their storage utilization in single-measure datasets. This requires properly utilizing the wavelet tree structure to identify these storage dependencies and processing the nodes in the wavelet tree using an appropriate ordering. Neither of these restrictions was present in [23].

4.3.1 Our Solution

We now formulate a dynamic programming (DP) solution for the optimization problem of Section 4.2; the notation used is shown in Table 4.3. Let $\mathrm{M}[i, B]$ denote the maximum benefit acquired when assigning at most space B to the subtree of coefficient c_i. Also, let $\mathrm{F}[i, B]$ denote the optimal benefit acquired when assigning at most space B to the subtree of coefficient c_i and when c_i is *forced* to be stored. Equation 4.1 depicts the recurrences employed by our HCDynL2 algorithm in order to calculate these values. Case 2, discussed above, corresponds to the first clause of the *max* calculation for $\mathrm{F}[i, B]$, while Case 3 is covered by the next two clauses of the same *max* calculation. Of course, when the remaining space is less than S_1 or $i \geq N$, it is infeasible [3] to store the coefficient value c_i, thus returning a benefit of $-\infty$. For the calculation of the $\mathrm{M}[i, B]$ value, Case 1 is covered in the first clause of the *max* quantity, while Cases 2 and 3 are covered in the second clause ($\mathrm{F}[i, B]$). Of course, if the remaining space is less than S_1 or $i \geq N$, no coefficient value can be stored, thus returning a benefit of 0 for $\mathrm{M}[i, B]$.

Given Equation 4.1, our HCDynL2 algorithm starts at the root of the wavelet tree and seeks to calculate the value of $\mathrm{M}[0, B]$. In this process, various M[] and F[] values are calculated. For each of these calculations we also record which clause of the formulas helped determine these values, and the corresponding allotments b_L to the left subtree of the nodes (see Equation 4.1). This step helps to quickly trace the steps of the algorithm when reconstructing the optimal solution.

After the value $\mathrm{M}[0, B]$ has been calculated, we can reconstruct the optimal

[3] Even though c_i can be stored for $S_2 \leq B < S_1$, there will be insufficient space ($< S_1$) to allocate to the lowest node of the path that c_i is attached to.

$$
\mathrm{F}[i,B] = \begin{cases} -\infty & \text{, if } i \geq N \\ & \text{or } B < S_1 - 1 \\ \max \left\{ \begin{array}{l} \max\limits_{0 \leq b_L \leq B-S_2-1} (c_i^*)^2 + \mathrm{G}_{[2i,b_L]} + \mathrm{M}_{[2i+1,B-b_L-S_2-1]} \\ \max\limits_{0 \leq b_L \leq B-S_2-1} (c_i^*)^2 + \mathrm{M}_{[2i,b_L]} + \mathrm{G}_{[2i+1,B-b_L-S_2-1]} \\ \max\limits_{0 \leq b_L \leq B-S_2} (c_i^*)^2 + \mathrm{F}_{[2i,b_L]} + \mathrm{M}_{[2i+1,B-b_L-S_2]} \\ \max\limits_{0 \leq b_L \leq B-S_2} (c_i^*)^2 + \mathrm{M}_{[2i,b_L]} + \mathrm{F}_{[2i+1,B-b_L-S_2]} \end{array} \right\} & \text{, otherwise} \end{cases}
$$

$$
\mathrm{G}[i,B] = \begin{cases} -\infty & \text{, if } i \geq N \\ & \text{or } B < S_1 - 1 \\ \max\limits_{0 \leq b_L \leq B-S_1+1} (c_i^*)^2 + \mathrm{M}_{[2i,b_L]} + \mathrm{M}_{[2i+1,B-b_L-S_1+1]} & \text{, otherwise} \end{cases}
$$

$$
\mathrm{M}[i,B] = \begin{cases} 0 & \text{, if } i \geq N \\ & \text{or } B < S_1 - 1 \\ \max \left\{ \begin{array}{l} \max\limits_{0 \leq b_L \leq B} \mathrm{M}_{[2i,b_L]} + \mathrm{M}_{[2i+1,B-b_L]} \\ \mathrm{F}_{[i,B]} \\ \mathrm{G}_{[i,B]} \end{array} \right\} & \text{, otherwise} \end{cases}
\tag{4.2}
$$

solution as follows. We start from the root node with a space constraint B. Based on which clause determined the value of $\mathrm{M}[i,B]$, we recurse to the two subtrees with the appropriate space allocation (recall that this information was recorded in the calculation of the M[] and F[] values) and a list of *hanging* coefficient values. These coefficient values belong to the hierarchically compressed path that passes through c_i. This path needs to be included in the recursion process because it can only be stored when all of its the coefficient values have been identified. Based on Cases 1-3 described above, at each node we may either: (i) Not store c_i; then store the input hanging path if it is non-empty; (ii) Attach c_i to the received hanging path (Case 2) and store the resulting hierarchically compressed coefficient; or (iii) Attach c_i to the received hanging path (Case 3) and recurse to the two subtrees. In this recursion, the resulting hanging path needs to be input to the appropriate subtree, while the other subtree will receive an empty hanging path.

Theorem 4.3.1. *The* HCDynL2 *algorithm computes the optimal* $\mathrm{M}[i,B]$ *and* $\mathrm{F}[i,B]$ *values at each node of the wavelet tree and for any space constraint* B *correctly.*

Proof. We will prove Theorem 4.3.1 by induction on the height of each coefficient from the bottom of the wavelet tree (i.e., leaf nodes correspond to height 1).

Base Case: Leaf nodes (height = 1). If coefficient c_i belongs at the leaf level, then the possible set of paths in the subtree of c_i degenerates to simply storing c_i. Thus, the optimal benefit of a solution $\mathrm{M}[i,B]$ is equal to $(c_i^*)^2$ for $B \geq S_1$ and 0, otherwise. Similarly, for $B \geq S_1$, $\mathrm{F}[i,B] = \mathrm{M}[i,B] = (c_i^*)^2$. Otherwise, c_i cannot be stored because of space constraints (thus the benefit is set $-\infty$ to represent this). Notice that in all cases the formulas for calculating $\mathrm{M}[i,B]$ and $\mathrm{F}[i,B]$ correctly compute the optimal solution and its benefit for any leaf node c_i and for any space

constraint B assigned to the subtree of the node.

Inductive Step. Assume that the HCDynL2 algorithm correctly computes the optimal $\mathrm{M}[i, B]$ and $\mathrm{F}[i, B]$ values at each node of the wavelet tree up to height j and for any space constraint B. We will show that the HCDynL2 algorithm also correctly computes the optimal $\mathrm{M}[i, B]$ (the proof for $\mathrm{F}[i, B]$ is similar) values at each node at height $j + 1$.

Note that the HCDynL2 algorithm considers all combinations of storing (or not) the root coefficient at each subtree and attaching this coefficient (or not) to optimal solutions calculated by the node's two subtrees. Thus, a case of sub-optimality may occur only if the optimal solution at node i needs to be computed by using a suboptimal solution (other than the computed $\mathrm{M}[2i, B]$ and $\mathrm{M}[2i + 1, B]$ values, or the $\mathrm{F}[2i, B]$ and $\mathrm{F}[2i + 1, B]$ values when c_i is stored) at (at least) one of its two subtrees.

Let the suboptimal solution needed to be considered is over a solution computed over the left subtree of c_i (i.e., the subtree of coefficient c_{2i}). Situations where the suboptimal solution is over the right subtree (or over both subtrees) are handled in a similar way.

First Case: $\mathrm{M}[i, B]$ does not store c_i. Consider that the optimal solution at a coefficient c_i that lies at height $j + 1$ of the wavelet tree for a space constraint B is produced by not storing c_i, but by considering solutions LSOL and RSOL at the left and right subtrees, respectively, of c_i with corresponding maximum space b_L and b_R. Let the solution $\mathrm{M}'[2i, b_L]$ at the left subtree be a suboptimal one, meaning that $\mathrm{M}'[2i, b_L] < \mathrm{M}[2i, b_L]$. Then, a solution that would consider RSOL and the solution of $\mathrm{M}[2i, b_L]$ requires at most space $b_L + b_R$ and has a larger benefit than the optimal solution of LSOL and RSOL. We therefore reached a contradiction.

Second Case: $\mathrm{M}[i, B]$ stores c_i and does not attach it in paths of the solutions of any subtree. In this case, if the optimal solution needs to consider a sub-optimal solution LSOL at the left subtree of c_i with space b_L, then obviously HCDynL2 examines the solution that stores $\mathrm{M}[2i, b_L]$ instead of LSOL, and which results in a larger benefit. We therefore reached a contradiction.

Third Case: $\mathrm{M}[i, B]$ stores c_i and attaches it to suboptimal solution LSOL (RSOL) at left (right) subtree. In this case, note that c_{2i} must be stored in the suboptimal solution LSOL (RSOL) considered at the left (right) subtree (and thus c_i requires space S_2 to be stored). Note that the solution that stores c_i and attaches it to the solution of $\mathrm{F}[2i, b_L]$ ($\mathrm{F}[2i + 1, b_R]$), where b_L (b_R) denotes the space of the suboptimal solution LSOL (RSOL), while also storing $\mathrm{M}[2i + 1, B - b_L - S_2]$ ($\mathrm{M}[2i, B - b_R - S_2]$) will result in a larger benefit, due to the inductive hypothesis. We therefore reached a contradiction.

□

4.3.2 Running Time and Space Complexities

Consider a node c_i at height j in the wavelet tree. Since there can be at most $2^j - 1$ coefficients below the subtree rooted at node c_i, the total budget allocated cannot exceed $2^j \cdot S_1$. Therefore, at any node c_i, HCDynL2 must calculate at most $\min\{B, 2^j S_1\}$ entries (if $2^j S_1 < B$, all space allotments larger than $2^j S_1$ result in the same benefit as that of the allotment for $2^j S_1$ and need not be computed), where

each requires time $\min\{B,\ 2^j S_1\}$ to consider all possible space allotments to the children nodes. Given that there are $N/2^j$ nodes at height j and summing across all $\log N$ heights we obtain (note that $B = 2^j S_1$ when $j = \log \frac{B}{S_1}$):

$$\sum_{j=1}^{\log N} \frac{N}{2^j} \left(\min\{B, 2^j S_1\}\right)^2 = \sum_{j=1}^{\log \frac{B}{S_1}} \frac{N}{2^j} 2^{2j} S_1^2 + \sum_{j=\log \frac{B}{S_1}+1}^{\log N} \frac{N}{2^j} B^2$$

$$= NS_1^2 \sum_{j=1}^{\log \frac{B}{S_1}} 2^j + NB^2 \sum_{j=\log \frac{B}{S_1}+1}^{\log N} \frac{1}{2^j}$$

$$= NS_1^2 \cdot O(\frac{B}{S_1}) + NB^2 \cdot O(\frac{S_1}{B}) = O(S_1 NB) = O(NB).$$

Note that the reconstruction process simply requires a top-down traversal of the wavelet tree. Therefore, the total running time remains $O(NB)$. Using similar arguments, we obtain that the space complexity is:

$$\sum_{j=1}^{\log \frac{B}{S_1}} \frac{N}{2^j} 2^j S_1 + \sum_{j=\log \frac{B}{S_1}+1}^{\log N} \frac{N}{2^j} B = NS_1 \log B + NB \cdot O(\frac{S_1}{B})$$

$$= O(N \log B).$$

Theorem 4.3.2. *The* HCDynL2 *algorithm constructs the optimal HCWS, given a space budget of B, in $O(NB)$ time using $O(N \log B)$ space.*

A Streaming Variant. The HCDynL2 algorithm can be easily modified to operate in one pass over the data using limited memory — i.e., in a data stream setting. Recall that HCDynL2 requires two passes over the data, one bottom-up for computing the optimal benefit while marking the decisions made, and another top-down for constructing the optimal HCWS. The streaming variation, denoted as HCDynL2-Str, makes two important observations: (i) not all the entries of the dynamic programming array are needed for the construction of the optimal HCWS; and (ii) in order to reconstruct the solution, the selected coefficients must be carried at each node along each M[] and F[] entry. Thus, in principle our HCDynL2-Str algorithm follows the same observations made in [43]. However, as explained later in this section, our main technical contribution in the HCDynL2-Str algorithm involves the ability to calculate the M[] and F[] entries and perform the memory cleanup in an efficient way, through some careful book-keeping. This step was not present in [43]. The following definition is helpful in our remaining discussion.

Definition 4.3.2. *A wavelet coefficient is termed as* closed *if we have observed all the data values in its support region. A wavelet coefficient is termed as* active *if it is not* closed *and for which we have already observed at least one data value in its support region. A wavelet coefficient is termed as* inactive *if it is neither active nor closed.*

For the first observation, notice that at any time a new data value d_i is read, then the *active* wavelet coefficients, which lie in path(d_i), need to be updated and their corresponding M[] and F[] entries need to be calculated again. Each such *active* node will also require in its operation the corresponding M[] and F[] entries of its child node that does not lie in path(d_i). Thus, for any space allotment b, only $O(\log N)$ (rather than $O(N)$) entries M$[\cdot, b]$ and F$[\cdot, b]$ need to be in memory, for any b — the remaining entries are only required for the second pass over the wavelet tree.

For the algorithm to operate in one pass, the price that has to be paid is that of increased space requirements per M[] and F[] entry. Namely, following the second observation, we need a factor of $O(\min\{B, 2^j\})$ more space to store the HCCs calculated so far at each node that belongs at height j of the wavelet tree (again, this space is required only for the aforementioned $O(\log N)$ active nodes). An important observation is that through some careful book-keeping for each entry M$[\cdot, b]$, F$[\cdot, b]$ we only require $O(1)$ time to calculate the HCCs involved. To achieve this, we maintain the selected HCCs at each of the aforementioned $O(\log N)$ nodes as a list, where the first element is always the HCC that includes the root coefficient of the node's subtree (note that such a HCC may not exist for the M[] entry). The selected HCCs of a node are updated when data values in its support region are observed. However, as we explain later in this section, for all nodes that become *closed* we need to perform a cleanup operation that removes from main memory certain HCCs of these nodes. This cleanup operation is thus performed only once, and not per observed data value in their support region.

For each allotment b to the node's subtree the HCCs for the M[] and F[] entries can be computed as follows (assuming that b_L (b_R) space is allocated to the node's left (right) subtree):

1. If c_i is stored and attached to a path originating from the node's left (right) subtree, then create a new HCC that is the result of adding c_i to the first HCC that corresponds to the solution of the F$[2i, b_L]$ (F$[2i+1, b_R]$). We then link this new HCC to the second element of the list of F$[2i, b_L]$ (F$[2i+1, b_R]$) and with the corresponding list of the M$[2i+1, b_R]$ (M$[2i, b_L]$) entry.

2. If c_i is stored but is not attached to any path originating from the node's left (right) subtree, then create a new HCC containing only c_i. Then link to this HCC the lists of HCCs that correspond to M$[2i, b_L]$ and M$[2i+1, b_R]$.

3. If c_i is not stored then simply link the lists of HCCs that correspond to M$[2i, b_L]$ and M$[2i+1, b_R]$.

All the above operations can be completed in $O(1)$ time, along with the removal of the HCCs that were created at node c_i (and linked to the HCC lists calculated at the children nodes of c_i) at the observation of previous data values (see the above 3 cases). Please note that when we compute the final list of HCCs for any node c_i (after the node becomes *closed*), then any HCCs of its children nodes c_{2i} and c_{2i+1} that are not part of the HCCs stored at node c_i are no longer needed and need to be deleted. This can be easily detected by examining how the M$[i, b]$ and F$[i, b]$ entries at node c_i were calculated. Assuming that c_i belongs at height j of the wavelet tree, this can be achieved in $O(\min\{B, 2^j\})$ time per each space allotment $b \leq B$ to c_i.

Thus, the overall running time requirements of the algorithm become (since the HCCs of each node are calculated continuously, but the memory cleanup is performed just once per node and per space allotment $b \leq B$):

$$\begin{aligned}
&\sum_{j=1}^{\log N} \frac{N}{2^j}\left(\left(\min\{B, 2^j\}\right)^2 + \log N \cdot \min\{B, 2^j\}\right) \\
&= \sum_{j=1}^{\log B} \frac{N}{2^j}(2^{2j} + 2^j \log N) + \sum_{j=\log B+1}^{\log N} \frac{N}{2^j}(B^2 + B \log N) \\
&= N \sum_{j=1}^{\log B}(2^j + \log N) + NB(B + \log N) \sum_{j=\log B+1}^{\log N} \frac{1}{2^j} \\
&= O(NB + N \log N \log B) + N(B + \log N) \\
&= O(NB + N \log N \log B).
\end{aligned}$$

To summarize, HCDynL2-Str operates in one pass over the data and gains in space by storing only $B \log N$ entries, which, however, each requires $O(B)$ space for the storage of its HCCs. Moreover, at any specific moment the currently selected HCWS can be accessed directly from the root node of the wavelet tree.

Theorem 4.3.3. *The* HCDynL2-Str *algorithm constructs the optimal HCWS in one pass, given a space budget of B, in $O(B + \log N \log B)$ amortized time per processed data value using $O(B^2 \log N)$ space.*

4.3.3 Achieved Benefit vs. Classic Method

A question that naturally arises is how does the benefit of the solution achieved by the HCDynL2 algorithm compare to the one achieved by a traditional technique (Classic) that individually stores the coefficients with the largest absolute normalized values. Consider a set $\mathcal{S} = S_1, \ldots, S_B$ of B stored coefficient values, sorted in non-increasing order of their absolute normalized values, by the traditional thresholding algorithm. Consider the case where these coefficient values lie in distant regions of the wavelet tree. Using a space constraint equal to $B \times (sizeof(Coord) + sizeof(Value)) = B \times (S_1 - 1)$ the benefit of the HCDynL2 algorithm cannot be smaller than the benefit of storing the first $m = \lfloor \frac{B \times (S_1 - 1)}{S_1} \rfloor$ coefficient values of $\mathcal{S}$ as hierarchically compressed wavelet coefficients, each storing exactly one coefficient value. Thus, in the worst case the ratio of benefits of the HCDynL2 algorithm over the Classic algorithm, as described above, may be as low as $\frac{Benefit(\text{HCDynL2})}{Benefit(\text{Classic})} = \frac{Benefit(S_1,\ldots,S_m)}{Benefit(S_1,\ldots,S_B)} \geq \frac{m}{B}$, since the coefficients in $\mathcal{S}$ are sorted.

On the other hand, the best case for the benefit of the HCDynL2 algorithm may occur for a storage constraint of $B' = S_1 + (\log N + 1) \times S_2$. In this case if the $\log N + 1$ coefficient values with the largest absolute normalized values lie on the same root-to-leaf path of the wavelet tree, then the ratio of benefits of the HCDynL2 algorithm over the Classic algorithm will be as high as (for $m' = \lfloor \frac{B'}{S_1 - 1} \rfloor$) : $\frac{Benefit(\text{HCDynL2})}{Benefit(\text{Classic})} = \frac{Benefit(S_1,\ldots,S_{\log N+1})}{Benefit(S_1,\ldots,S_{m'})} \leq \frac{\log N+1}{m'}$. Therefore, the following theorem holds.

Theorem 4.3.4. *The* HCDynL2 *algorithm, when compared to the* Classic *algorithm, given the same space budget, exhibits a benefit ratio of*

$$\frac{\left\lfloor B \times \frac{S_1-1}{S_1} \right\rfloor}{B} \leq \frac{Benefit(\mathsf{HCDynL2})}{Benefit(\mathsf{Classic})} \leq \frac{\log N + 1}{\left\lfloor \frac{S_1+(\log N+1)S_2}{S_1-1} \right\rfloor}.$$

An Improved-Benefit Variant. It is important to emphasize that the HCDynL2 algorithm can be easily modified to guarantee that its produced solution has a benefit at least equal to the one of the traditional approach. This can be achieved by allowing HCCs with a single stored coefficient value to drop the very small overhead of the single bit and be stored in a separate storage. In this case, the first stored coefficient in a HCC requires space $S_1 - 1$, the second coefficient value in the same HCC requires additional space equal to $S_2 + 1$, while any additional coefficient values in the same HCC require space S_2 to be stored. This results in constructing a modified HCWS* synopsis.

The main difference of the modified algorithm, denoted as HCDynL2*, compared to the discussion of Section 4.3 is that now, due to the different space needed for the second and third coefficient values of each HCC, two suboptimal solutions need to be maintained (see Equation 4.2): (i) $\mathrm{F}[i, B]$, the benefit of the optimal solution when assigning space at most equal to B to the subtree of coefficient c_i and when both c_i and one of its children (c_{2i} or c_{2i+1}) are stored; and (ii) $\mathrm{G}[i, B]$, the benefit of the optimal solution when assigning space at most equal to B to the subtree of coefficient c_i and c_i is stored as the bottom-most coefficient in a path. Note, that for the second suboptimal solution the space required is $S_1 - 1$, as discussed. For the first suboptimal solution two cases exist: (a) c_i is the second coefficient in a path, hence, the space required is $S_2 + 1$ and further, a suboptimal solution G[] in one of its children must be combined with an optimal solution M[] in the other child (the first two non-trivial cases of $\mathrm{F}[i, B]$ in Equation 4.2); and (b) c_i is not the second coefficient (it could be the third or more), hence, the space required is S_2 and further, a suboptimal solution F[] in one of its children must be combined with an optimal solution M[] in the other child (the last two non-trivial cases of $\mathrm{F}[i, B]$ in Equation 4.2. Therefore, the following theorem holds.

Theorem 4.3.5. *The* HCDynL2* *algorithm constructs a modified HCWS* synopsis such that the obtained benefit is never less than that of the* Classic *algorithm, given the same space budget:*

$$Benefit(\mathsf{HCDynL2}^*) \geq Benefit(\mathsf{Classic}).$$

It is important to note that the asymptotic running time and space requirements of the HCDynL2* algorithm are the same as those of the HCDynL2 algorithm. However, since its implementation requires the evaluation of three DP recurrences, its actual running time and space requirements are about 50% increased over the ones of HCDynL2. Finally, a streaming variant of the HCDynL2* algorithm can be obtained in a manner analogous to that of HCDynL2. Similarly, an approximation algorithm for the HCDynL2* algorithm can be obtained in a manner analogous to the approximation algorithm of HCDynL2 (presented in Section 4.4).

$$\left.\begin{aligned}
\textsc{apprF}[i, p_j^L + p_k^R + S_1] &= (c_i^*)^2 + \textsc{apprM}[2i, p_j^L] + \textsc{apprM}[2i+1, p_k^R] \\
\textsc{apprF}[i, q_j^L + p_k^R + S_2] &= (c_i^*)^2 + \textsc{apprF}[2i, q_j^L] + \textsc{apprM}[2i+1, p_k^R] \\
\textsc{apprF}[i, p_j^L + q_k^R + S_2] &= (c_i^*)^2 + \textsc{apprM}[2i, p_j^L] + \textsc{apprF}[2i+1, q_k^R]
\end{aligned}\right\} \tag{4.3}$$

$$\left.\begin{aligned}
\textsc{apprM}[i, p_j^L + p_k^R + S_1] &= \textsc{apprF}[i, p_j^L + p_k^R + S_1] \\
\textsc{apprM}[i, q_j^L + p_k^R + S_2] &= \textsc{apprF}[i, q_j^L + p_k^R + S_2] \\
\textsc{apprM}[i, p_j^L + q_k^R + S_2] &= \textsc{apprF}[i, p_j^L + q_k^R + S_2] \\
\textsc{apprM}[i, p_j^L + p_k^R] &= \textsc{apprM}[2i, p_j^L] + \textsc{apprM}[2i+1, p_k^R]
\end{aligned}\right\} \tag{4.4}$$

Symbol	Description
$\textsc{apprM}[i, x]$	Approximate value for optimal benefit when assigning at most space x to the subtree rooted at coefficient c_i
$\textsc{apprF}[i, x]$	Approximate value for optimal benefit when assigning at most space x to the subtree rooted at coefficient c_i and when c_i is forced to be stored
$\{p_j^i\}$	Set of breakpoints for $\textsc{apprM}[i, \cdot]$
$\{q_k^i\}$	Set of breakpoints for $\textsc{apprF}[i, \cdot]$
ϵ	Approximation factor
δ	Degradation factor incurred at each level

Table 4.4: *Notation used in* HCApprL2 *Algorithm*

4.4 HCApprL2: An Approximation Algorithm

In this section we propose an approximation algorithm for efficiently constructing hierarchically compressed wavelet synopses. Our algorithm, termed HCApprL2, offers significant improvements in time and space requirements over HCDynL2 while providing tunable error guarantees. The HCApprL2 algorithm constructs a HCWS that has a benefit that does not exceed the optimal synopsis, but definitely not less than $\frac{1}{1+\epsilon}$ of the optimal benefit, for some given parameter ϵ. Clearly, smaller values for ϵ lead to more accurate synopses; HCApprL2 solves Problem 4.2 optimally for $\epsilon = 0$.

The HCApprL2 algorithm constructs functions $\textsc{apprM}[]$, $\textsc{apprF}[]$ and computes their values at some space allotment in a similar manner to how HCDynL2 computes M[] and F[] values (i.e., the values at a non-leaf node depend on the values of its children) but does so for a sparse set of space allotments, termed breakpoints, rather than for all possible allotments.

The HCApprL2 algorithm operates on the wavelet tree in a bottom-up manner. At each node it creates a set of *candidate* breakpoints by combining breakpoints from the children of the node. Then, in a two-phase trimming process it eliminates some of these candidates to obtain the actual breakpoints of the node. This trimming process is responsible for bounding the error incurred by not examining all space allotments, as it will become apparent in the next section.

4.4.1 Breakpoint Calculation

The crux of the HCApprL2 algorithm lies in the calculation of the breakpoints and their corresponding benefit values for functions $\textsc{apprM}[]$ and $\textsc{apprF}[]$ at each node. The algorithm proceeds in a bottom-up manner, starting from the leaf nodes at

height 1.

Assume that node c_i, at height 1, is a non-zero leaf coefficient. In this case there are two breakpoints 0 and S_1 with approximate benefits 0 and $(c_i^*)^2$ respectively for both approximation functions. In the case of a zero valued leaf coefficient $\text{APPRM}[i,\cdot]$ has only breakpoint 0 with zero benefit, whereas $\text{APPRF}[i,\cdot]$ has breakpoints $0, S_1$ with benefits $-\infty$ and 0 respectively.

For all non-leaf nodes the breakpoint calculation proceeds following the same steps: (i) a set of candidate breakpoints is created by combining all breakpoints of the children; (ii) a trimming process reduces this set to the actual breakpoints to be used as input for the first step in the parent node.

Consider a non-leaf node c_i at height l; since HCApprL2 proceeds bottom up all breakpoints for nodes lower in the tree have been calculated. Thus, let $\{p_j^L\}$, $\{q_k^L\}$ denote the set of breakpoints for $\text{APPRM}[2i,\cdot]$ and $\text{APPRF}[2i,\cdot]$ functions for the left child of c_i and let $\{p_j^R\}$, $\{q_k^R\}$ denote the set of breakpoints for $\text{APPRM}[2i+1,\cdot]$ and $\text{APPRF}[2i+1,\cdot]$ functions for the right child of c_i.

The candidate breakpoints for $\text{APPRF}[i,\cdot]$ and the corresponding benefit values are calculated combining all breakpoints from sets $\{p_j^L\}$, $\{q_k^L\}$, $\{p_j^R\}$, $\{q_k^R\}$ as shown in Equation 4.3 — candidate breakpoints of space more than B are easily identified and rejected. Observe that these equations correspond to the non-trivial cases of the defining recurrence for $\text{F}[i,\cdot]$ (Equation 4.1). The algorithm considers the following cases for all j,k:

- Store c_i using space S_1 and combine all (approximately) optimal solutions $\text{APPRM}[2i, p_j^L]$, $\text{APPRM}[2i+1, p_j^R]$.
- Store c_i using space S_2 and combine all (approximately) optimal when forced to store c_{2i} solutions $\text{APPRF}[2i, q_k^L]$ with all (approximately) optimal solutions $\text{APPRM}[2i+1, p_j^R]$.
- Store c_i using space S_2 and combine all (approximately) optimal solutions $\text{APPRM}[2i, p_j^L]$ with all (approximately) optimal when forced to store c_{2i+1} solutions $\text{APPRF}[2i+1, q_k^R]$.

Similarly, the candidate breakpoints for $\text{APPRM}[i,\cdot]$ and their corresponding benefit values are also calculated combining all breakpoints from sets $\{p_j^L\}$, $\{q_k^L\}$, $\{p_j^R\}$, $\{q_k^R\}$ as shown in Equation 4.4 — candidate breakpoints of space more than B are easily identified and rejected. Again, observe that these equations correspond to the non-trivial cases of the defining recurrence for $\text{M}[i,\cdot]$ (see Equation 4.1), which in addition to the candidate breakpoints considered for $\text{APPRF}[i,\cdot]$ considers the following case, for all j,k: Do not store c_i and combine all (approximately) optimal solutions, i.e., the pairs $\text{APPRM}[2i, p_j^L]$, $\text{APPRM}[2i+1, p_j^R]$.

Once all candidate breakpoints have been calculated we perform a two-phase trimming process for each approximation function, to reduce the number of breakpoints.

First Phase. We remove the useless configurations — those that cost more but have less benefit than others. This can be done by a simple ordering of the configurations increasingly by their approximate benefit values and a subsequent linear scan.

Second Phase. The final breakpoints $\{p_j^i\}$, $\{q_k^i\}$ are set as follows. Consider the case of the approximate benefit function $\text{APPRM}[i,\cdot]$. Set p_1^i equal to the first candidate breakpoint (after sorting); it is easy to see that this breakpoint always

corresponds to space 0. The rest of the breakpoints are discovered iteratively: assuming breakpoint p^i_{k-1} has been found, breakpoint p^i_k is the smallest candidate breakpoint such that $\textsc{apprM}[i, p^i_k] > (1+\delta)\textsc{apprM}[i, p^i_{k-1}]$, for some parameter δ which depends on the desired approximation factor ϵ and whose value will be determined later in this section.

The following lemmas are a direct result of the trimming process.

Lemma 4.4.1. *For any node that belongs at height j of the wavelet tree, there can be at most $R_j = O\left(\min\{B, 2^j, \frac{1}{\delta}\log||W_A||\}\right)$ breakpoints.*

Proof. Certainly, there can be no more than B breakpoints for each approximation function. Similarly, since there can be at most $2^j - 1$ coefficients in the subtree rooted at each node that belongs at height j, the total number of space entries, and thus breakpoints, cannot exceed $2^j S_1 = O(2^j)$. Additionally, there can be no more than $\log_{1+\delta} \mathrm{M}[i, B]$ breakpoints for $\textsc{apprM}[i, \cdot]$ (and no more than $\log_{1+\delta} \mathrm{F}[i, B]$ for $\textsc{apprF}[i, \cdot]$), as $\mathrm{M}[i, B]$ (resp. $\mathrm{F}[i, B]$) is the highest possible benefit that can be attained at node c_i for space B. Since this benefit cannot be more than the energy of the wavelet transform $||W_A||^2$, the lemma easily follows for small δ values. □

Lemma 4.4.2. *Let $\{p^i_j\}$ be the set of breakpoints for approximation benefit function $\textsc{apprM}[i, \cdot]$. If b is a candidate breakpoint such that $b \in [p^i_k, p^i_{k+1})$, then $\textsc{apprM}[i, b] \leq (1+\delta)\cdot\ \textsc{apprM}[i, p^i_k]$ — i.e., b is covered by p^i_k within a $(1+\delta)$ degradation. Analogous result holds for function $\textsc{apprF}[]$.*

Proof. If b is not discarded in the first phase of the trimming process it is straightforward to see that the lemma holds. Now, assume that b was discarded in the first phase. Therefore, there must exist a candidate breakpoint $b' < b$ not discarded in the first phase with $\textsc{apprM}[i, b'] \geq \textsc{apprM}[i, b]$ such that b' is the highest non-discarded breakpoint smaller than b. Observe that b' and b are covered by the same breakpoint p^i_k (b' might be the breakpoint p^i_k): $b, b' \in [p^i_k, p^i_{k+1})$ and that the lemma holds for b'. Therefore, $\textsc{apprM}[i, b] \leq \textsc{apprM}[i, b'] \leq (1+\delta)\textsc{apprM}[i, p^i_k]$ and the lemma holds for b. □

By aggregating the degradation occurred at all descendants of a node we obtain the following.

Lemma 4.4.3. *Assume node c_i is at height h of the wavelet tree (equivalently at level $\log N - h$), and let $\{p^i_j\}$ and $\{q^i_j\}$ be the set of breakpoints for $\textsc{apprM}[i, \cdot]$ and $\textsc{apprF}[i, \cdot]$ respectively. Also let x, y be some arbitrary space allotments and let breakpoints p^i_k, q^i_k be such that $x \in [p^i_k, p^i_{k+1})$ (or $x \geq p^i_k$, if p^i_k is the last breakpoint) and $y \in [q^i_k, q^i_{k+1})$ (or $y \geq q^i_k$, if q^i_k is the last breakpoint). The approximate benefit value computed at node c_i (the approximate benefit value when c_i is forced to be stored) for space p^i_k (resp. q^i_k) is not less than $\frac{1}{(1+\delta)^{h-1}}$ of the optimal value (resp. optimal value when c_i is forced to be stored). That is, $\mathrm{M}[i, x] \leq (1+\delta)^{h-1}\textsc{apprM}[i, p^i_k]$ and $\mathrm{F}[i, y] \leq (1+\delta)^{h-1}\textsc{apprF}[i, q^i_k]$.*

Proof. We prove the lemma for $\textsc{apprF}[i, \cdot]$ and $\textsc{apprM}[i, \cdot]$, by induction on the height h of the wavelet tree node c_i belongs to. The base case $h = 1$ holds by construction: Assume coefficient c_i is non-zero; thus only breakpoints $p^i_1 = 0$, $p^i_2 = S_1$ and $q^i_1 = 0$, $q^i_2 = S_1$ exist for approximation functions $\textsc{apprM}[i, \cdot]$ and $\textsc{apprF}[i, \cdot]$ respectively. Clearly, (i) when $x \in [p^i_1, p^i_2)$, $\textsc{apprM}[i, p^i_1] = \mathrm{M}[i, x]$; (ii) when $y \in$

$[q_1^i, q_2^i)$, $\textsc{apprF}[i, q_1^i] = \mathrm{F}[i, y]$; (iii) when $x \geq p_2^i$, $\textsc{apprM}[i, p_2^i] = \mathrm{M}[i, x]$; and (iv) when $y \geq q_2^i$, $\textsc{apprF}[i, q_2^i] = \mathrm{F}[i, y]$. In the case of a zero valued coefficient c_i, only breakpoint p_1^i exists and the reasoning is similar.

Assuming the hypothesis holds for all nodes at height h we will show that it holds for nodes at height $h+1$. We will only consider the approximation function $\textsc{apprF}[i, \cdot]$ for node c_i at height $h+1$, as the proof for $\textsc{apprM}[i, \cdot]$ is similar. Further, assume that the optimal benefit $\mathrm{F}[i, y]$ when c_i is forced to be stored for a space budget of y is constructed from the second non-trivial clause of Equation 4.1 by allotting space S_2 to coefficient c_i, y_L to the left subtree and $y - y_L - S_2$ to the right subtree; that is, $\mathrm{F}[i, y] = (c_i^*)^2 + \mathrm{F}[2i, y_L] + \mathrm{M}[2i+1, y - y_L - S_2]$. The proof is similar for the other clauses and thus omitted.

If $\{q_j^L\}$ and $\{p_j^R\}$ denote the sets of breakpoints for functions $\textsc{apprF}[2i, \cdot]$ and $\textsc{apprM}[2i+1, \cdot]$ respectively, let q_k^L be the highest breakpoint not exceeding y_L and let p_k^R be the highest breakpoint less than $y - y_L - S_2$. By the induction hypothesis

$$\mathrm{F}[2i, y_L] \leq (1+\delta)^{h-1} \textsc{apprF}[2i, q_k^L] \quad \text{and}$$
$$\mathrm{M}[2i+1, y - y_L - S_1] \leq (1+\delta)^{h-1} \textsc{apprM}[2i+1, p_k^R].$$

Define $b = q_k^L + p_k^R + S_1$. Certainly, b was a candidate breakpoint for function $\textsc{apprF}[i, \cdot]$ and considered by our HCApprL2 algorithm (see Equation 4.3):

$$\textsc{apprF}[i, b] = (c_i^*)^2 + \textsc{apprF}[2i, q_k^L] + \textsc{apprM}[2i+1, p_k^R].$$

Using the above equation and the induction hypothesis, optimal value $\mathrm{F}[i, y]$ is bounded as follows.

$$\begin{aligned} \mathrm{F}[i, y] &= (c_i^*)^2 + \mathrm{F}[2i, y_L] + \mathrm{M}[2i+1, y - y_L - S_2] \\ &\leq (c_i^*)^2 + (1+\delta)^{h-1} \left(\textsc{apprF}[2i, q_k^L] + \textsc{apprM}[2i+1, p_k^R] \right) \\ &\leq (1+\delta)^{h-1} \textsc{apprF}[i, b] \end{aligned}$$

Now, either b belongs to $[q_k^i, q_{k+1}^i)$ or not. Consider the first case. By Lemma 4.4.2 $\textsc{apprF}[i, b] \leq (1+\delta)\textsc{apprF}[i, q_k^i]$ and thus:

$$\mathrm{F}[i, y] \leq (1+\delta)^{h-1} \textsc{apprF}[i, b] \leq (1+\delta)^h \textsc{apprF}[i, q_k^i].$$

In the other case, observe that b must be smaller than q_k^i, because $b \leq y \in [q_k^i, q_{k+1}^i)$. Therefore, since $\textsc{apprF}[i, b] \leq \textsc{apprF}[i, q_k^i]$:

$$\mathrm{F}[i, y] \leq (1+\delta)^{h-1} \textsc{apprF}[i, b] \leq (1+\delta)^{h-1} \textsc{apprF}[i, q_k^i].$$

Thus, in either case $\mathrm{F}[i, y] \leq (1+\delta)^h \textsc{apprF}[i, q_k^i]$. □

Finally, we obtain the following.

Theorem 4.4.1. *The* HCApprL2 *algorithm provides a HC synopsis to Problem 4.2 that needs space not more than B and has benefit not less than $\frac{1}{1+\epsilon}$ of the optimal benefit. Assuming p_k^0 is the highest breakpoint of function $\textsc{apprM}[0, \cdot]$ not exceeding B, we have $\mathrm{M}[0, B] \leq (1+\epsilon)\textsc{apprM}[0, p_k^0]$.*

Proof. Apply Lemma 4.4.3 for $\mathrm{M}[0, B]$ setting $\delta = \frac{\epsilon}{\log N}$.

□

4.4.2 Space and Running Time Complexities

The space and time complexity of the HCApprL2 algorithm depend on the number of breakpoints R_j (rather than solely on B) for each approximation function at each node that belongs at height j of the wavelet tree. Lemma 4.4.1 provides a bound for this number, if one sets $\delta = \frac{\epsilon}{\log N}$: $R_{max} = O\left(\min\{B, 2^j, \ \frac{1}{\epsilon}\log N \log ||W_A||\}\right)$.

At each node and for each approximation function, the HCApprL2 algorithm first computes candidate breakpoints by combining all breakpoints from the children nodes (in $O(R_j^2)$ time and space), sorts them (in $O(R_j^2 \log R_j)$ time) and performs the trimming process (in $O(R_j^2)$ time and space). Thus, the time requirement is $O(R_j^2 \log R_j)$ per node at height j of the wavelet tree. HCApprL2 requires a temporary space of $O(R_j^2)$ to perform the trimming process, but registers only $O(R_j)$ space per node. Using similar reasoning with the complexity analysis of the HCDynL2 algorithm we derive the following running time complexity our algorithm (by setting $K = \min\{B, \frac{1}{\epsilon}\log N \log ||W_A||\}$ - observe the time requirement increases by a factor of $\log R_j$ due to the sorting involved during the breakpoint calculation):

$$
\begin{aligned}
&O(\sum_{j=1}^{\log N} \frac{N}{2^j}\left(\left(\min\{2^j, K\}\right)^2 \log \min\{2^j, K\}\right)) = \\
&O(\sum_{j=1}^{\log K} \frac{N}{2^j} j 2^{2j} + \sum_{j=\log K+1}^{\log N} \frac{N}{2^j} K^2 \log K) \\
&= O(N \sum_{j=1}^{\log K} j 2^j + NK^2 \log K \sum_{j=\log K+1}^{\log N} \frac{1}{2^j}) \\
&= O(NK \log K).
\end{aligned}
$$

Using a similar calculation for the space requirements of the algorithm, the following Theorem easily follows.

Theorem 4.4.2. *Given space budget B, the* HCApprL2 *algorithm constructs a HCWS, in $O(NK \log K)$ time using $O(N \log K)$ space, where $K = \min\{B, \frac{1}{\epsilon}\log N \log ||W_A||\}$. The streaming variant of this algorithm requires only $O(K^2 \log N)$ space.*

Note that the streaming variant of the algorithm is analogous to the corresponding variant of the optimal DP algorithm, and is thus omitted from our presentation.

4.5 HCGreedyL2: A Greedy Heuristic

Due to the large space and running time requirements of the HCDynL2 and HCApprL2 algorithms, we now seek to devise a more efficient greedy solution for the same optimization problem. At first sight our optimization problem looks similar to the classical knapsack problem. However, our optimization problem is much more difficult for two reasons. First, even though the benefit of including any given coefficient in the synopsis is fixed, its space requirement depends on the position of the coefficient it the hierarchical path; it may require either S_1 or S_2 space. Second, considering the search space of all possible HCCs, observe that once a HCC is chosen, there is a large number of HCCs which become invalid and cannot be part of the solution; these are

Symbol	Description
GrM_i	Non-stored candidate path in c_i's subtree with the estimated maximum per space benefit
GrF_i	Non-stored candidate path in c_i's subtree with the estimated maximum per space benefit when storing c_i
$Owner_i$	The hierarchically compressed coefficient in which c_i belongs to (Ø if c_i has not been stored)
$GrM_i.b$	Benefit of GrM_i
$GrM_i.sp$	Needed space for GrM_i
$GrF_i.b$	Benefit of GrF_i
$GrF_i.sp$	Needed space for GrF_i
$State_i[0..2]$	Bitmap of node i, consisting of 3 bits: - If $State(0)$ is set, c_i has already been selected for storage - If $State(0)$ and $State(1)$ are set, c_i = bottom($Owner_i$) - Otherwise, if $State(0)$ is set, $State(2)$ denotes if set (not set) that c_i is part of a path through its left (right) subtree
chM_i, chF_i	2-bit bitmaps for retracing the algorithm choices (determine through which action the paths GrM_i and GrF_i were formed)

Table 4.5: *Notation used in* HCGreedyL2 *algorithm*

the hierarchically compressed coefficients that overlap with the chosen HCC. This dependency amongst the candidate HCCs is not typical in knapsack-like problems for which there exist greedy algorithms with tight approximation bounds.

In analogy to most greedy heuristics for knapsack-like problems, we try to formulate candidate solutions and utilize a per-space benefit heuristic at each step of the algorithm. In particular, our proposed HCGreedyL2 algorithm greedily allocates its available space by continuously selecting (until the space budget is exhausted) for storage the candidate path that (i) does not overlap any of the already selected for storage paths; and (ii) is estimated to exhibit the largest *per space benefit*, if included in the solution. To increase the effectiveness of the algorithm, it is crucial that, whenever possible, candidate paths be combined with paths already selected for storage, and that such storage dependencies be exploited. As we will explain shortly, this can be achieved by some careful book-keeping.

The operation of the algorithm is based on two main steps, that are repeated several times, and that we will detail shortly: (i) Selecting good candidate paths per subtree; and (ii) Marking candidate paths for storage and properly adjusting the benefits of non-stored candidate paths. The first of these phases first occurs at the initialization phase of the algorithm by visiting all the nodes of the wavelet tree, in order to setup the values of several variables at each node. Table 4.5 provides a synopsis of these variables, and of the notation used in the entire HCGreedyL2 algorithm. Appropriate definitions will be provided in our discussion whenever necessary. After this initialization phase, the coefficients in the path that is estimated at the root node to exhibit the best per space benefit are visited and marked for inclusion in the final solution (by modifying the *State* bitmap of these nodes). This is achieved by the second phase. Following each such marking process, the first phase needs to be called for each visited node of the wavelet tree. Observe that calls to the second phase and all subsequent calls to the first phase only visit nodes in the currently selected path.

Before proceeding to our discussion, it is important to emphasize that the paths GrM_i, GrF_i and $Owner_i$ (referenced in Table 4.5) are not stored at each node, but can rather be reconstructed by an appropriate traversal of the wavelet tree.

4.5.1 Candidate Path Selection

The computation of the best candidate path in a subtree of the wavelet tree structure is a bottom-up procedure. At each step of the algorithm, at each node c_i of the wavelet tree we store the benefit and the corresponding space of two candidate paths: (i) the candidate path GrM_i in the subtree of c_i that is estimated to achieve, if stored, the best per space benefit; and (ii) the candidate path GrF_i of c_i's subtree that is estimated to achieve the best per space benefit while storing the coefficient c_i. This implies that GrM_i might be any path of the subtree rooted at c_i, whereas GrF_i has to be a path containing c_i.

In order to compute these two candidate paths along with their corresponding benefits and their needed space, the HCGreedyL2 algorithm considers combining the coefficient value c_i with the candidate paths computed at c_i's two subtrees. This process utilizes some information that is produced during the operation of the algorithm and is stored as a bitmap *State* in each node, whereas the choices made are stored in *chF*, *chM* (see Table 4.5).

In the following, we omit discussion on what happens in the case of the root node for exposition purposes; the required changes due to the root having a single child are straightforward.

Computing GrF_i. The computation of GrF_i depends on whether c_i has been stored (i.e., whether $State_i(0)$ is set).

Coefficient c_i has been stored. In this case there is no candidate path that can store c_i. Thus, in this case we have $GrF_i = \emptyset$ and we set $GrF_i.b = GrF_i.sp = 0$ and $chF_i = 00$.

Coefficient c_i has not been stored. The following choices should be considered and the one with the highest per space benefit is selected (by appropriately setting the value of chF_i):

1. Storing simply c_i ($chF_i = 01$). The space requirements of this solution depends on whether c_i can be attached to an already selected path. If c_i is a non-leaf node in the wavelet tree and either $State_{2i}(0)$ or $State_{2i+1}(0)$ is set, then c_i can be attached to such a path (through the corresponding subtree) and $GrF_i.sp = S_2$. Otherwise, we set $GrF_i.sp = S_1$. This solution has a benefit equal to $(c_i^*)^2$ if the available space (at the step of the algorithm when this computation is performed) is at least $GrF_i.sp$, or 0 otherwise.
2. Storing c_i and combining it with GrF_{2i} ($chF_i = 10$) (or combining it with GrF_{2i+1} (chF_i=11)). This solution has an overall space requirement of S_2+$GrF_{2i}.sp$ (resp., S_2+ $GrF_{2i+1}.sp$) and a benefit of $(c_i^*)^2$+$GrF_i.b$ (resp., $(c_i^*)^2$ + $GrF_{2i+1}.b$) if the available space (at the step of the algorithm when this computation is performed) is at least $GrF_i.sp$, or $-\infty$ otherwise.

Moreover, in all three cases presented above, we decrease the $GrF.sp$ values by $S_1 - S_2$ if the parent node of c_i has been marked for storage and is also the bottom-most coefficient in its HCC. This is because GrF_i can help reduce, if selected for storage, the storage overhead for the parent node of c_i.

Computing GrM_i The computation of GrM_i also depends on whether c_i has been stored (i.e., whether $State_i(0)$ is set).

Coefficient c_i has not been stored. The following choices should be considered

Node	$GrM_i.b$	$GrM_i.sp$	$GrF_i.b$	$GrF_i.sp$	$State_i$	chM_i	chF_i
8	288	65	288	65	000	11	01
9	2	65	2	65	000	11	01
10	0	65	0	65	000	11	01
11	242	65	242	65	000	11	01
12	0	65	0	65	000	11	01
13	0	65	0	65	000	11	01
14	0	65	0	65	000	11	01
15	648	65	648	65	000	11	01
4	288	65	292	98	000	01	10
5	242	65	342	98	000	10	11
6	0	65	0	65	000	11	01
7	648	65	972	98	000	10	11
2	584	131	584	131	000	11	11
3	648	65	1134	131	000	10	11
1	3844	65	3844	65	000	11	01
0	10244	98	10244	98	000	11	11

Table 4.6: *Computed values at initialization phase*

and the one with the highest per space benefit is selected (by appropriately setting the chM_i value):

1. The candidate path of solution GrF_i ($chM_i = 11$).
2. For non-leaf nodes, GrM_i copies a candidate path from one of its children, either GrM_{2i} (chM_i=01) or GrM_{2i+1} (chM_i=10), selecting the one with the highest per space benefit.

Coefficient c_i has been stored. If c_i is a leaf node, then $GrM_i = \emptyset$ and we set $GrM_i.b = GrM_i.sp = 0$ and $chM_i = 00$. For non-leaf nodes, GrM_i examines the candidate paths GrM_{2i} and GrM_{2i+1} from its children nodes and copies the one that exhibits the largest per space benefit.

Example 4.5.1. *In Table 4.6 we depict the calculated GrM_i, GrF_i, $State_i$, chM_i and chF_i values and bitmaps computed at each node of Figure 4.1 during the initialization phase of the* HCGreedyL2 *algorithm. Based on the normalized coefficient values presented in Section 4.2, the benefit of storing each of the* 16 *coefficients is:* [6400, 3844, 242, 162, 4, 100, 0, 324, 288, 2, 0, 242, 0, 0, 0, 648]. *In this example, the size required to store a coefficient coordinate or a coefficient value has been set to 32 bits. The nodes in Table 4.6 have been ordered according to their resolution level. Details on the selected HCCs are provided later in this section. However, it is interesting to note that, even though the final selection of the HCCs has been presented in Section 4.2, the stored HCCs are produced by successive steps where smaller HCCs are merged. For example, by examining the GrF_1 values we observe that the HCC that is estimated to achieve the best per space benefit at node c_1 while also storing c_1 contains only the node c_1, and not the entire path c_{15}, c_7, c_3, c_1. This path will gradually be formed by the algorithm.*

4.5.2 Marking Paths for Storage

After the path with the overall per space benefit has been estimated (GrM_0), and its space $GrM_0.sp$ is subtracted from the available space, the process of traversing the wavelet tree to mark the coefficients in GrM_0 for storage is simple, due to the storage of the chM and chF bitmaps at each node. This top-down recursive process

starts at the root node and descends the path that leads to the node bottom(GrM_0). The steps of this process are:

1. At each node c_i of this path, we are asked to reconstruct either the path GrM_i or the path GrF_i. Notice that reconstructing GrM_i may lead to reconstructing GrF_i if $chM_i = 11$.
2. This process will never visit a node where the corresponding chM_i or chF_i values are equal to '00'.
3. If reconstructing GrF_i and $chF_i = 01$, then c_i is marked as stored by setting the bit $State_i(0)$ to 1. If in this case $GrF_i.sp = S_1$, then $State_i(1)$ is set and we recomputed the GrF and GrM values at the two children nodes of c_i, as described in Section 4.5.1. Otherwise, we reset $State(1)$ and assign the value of $State_i(2)$ depending on which path c_i can be attached to (if it can be attached to paths from both subtrees, pick any one of them randomly).
4. If reconstructing GrF_i and $chF_i = 10$ (11), we mark c_i for storage by setting $State_i(0)$ to 1, resetting the value of $State_i(1)$ and setting the value of $State_i(2)$ to 1 (0, respectively). We also recurse to reconstruct GrF_{2i} (GrF_{2i+1}).
5. If reconstructing GrM_i and $chM_i = 01$ (10), then we recurse to reconstruct GrM_{2i} (GrM_{2i+1}). After this recursion, we need to check if the newly stored path in the subtree of c_{2i} (c_{2i+1}) can be attached to c_i. By following the process described in Section 4.5.1, if this is detected the value of $State_i(1)$ is reset and the value of $State_i(2)$ is set to 1 (0, correspondingly). Also, in this case, the GrF and GrM values of c_{2i+1} (c_{2i}) need to be recalculated, since any path containing c_{2i+1} (c_{2i}) cannot lower, any more, the storage cost of c_i.
6. After possibly recursing to solutions in subtrees of c_i, the algorithm needs to recalculate the values of GrM_i and GrF_i, and all the corresponding chM_i and chF_i variables by executing the Candidate Path Selection phase on the visited nodes.

The only detail that we have not discussed is what happens if the selected path does not fit within the remaining space budget. In this case we simply traverse the selected path but mark for inclusion in the final solution only the highest coefficients in the path, such that the space constraint is not violated (we thus omit coefficients at the bottom of the path).

Example 4.5.2. *Returning our attention to Table 4.6, we notice that based on the chM_0 and chF_0 bitmaps, the selected solution will need to store the coefficient c_0 and combine it with an HCC at its subtree (since chF_0=11). The bit $State_0(0)$ is thus set, while the bit $State_0(1)$ remains unset since this coefficient will surely not be the bottom-most coefficient in its HCC. Since node 0 has only one child node in the wavelet tree, we must decide whether to consider that node 1 lies in its left or right subtree. We have selected the latter option and, thus, do not set the $State_0(2)$ bit. By recursing at node 1, we see based on the chM_1 and chF_1 bitmaps, that the coefficient c_1 needs to be stored, and that we do not need to recurse to children nodes. In this case, the bits $State_1(0)$ and $State_1(1)$ need to be set. Since c_1 became a new bottom-most coefficient at a new HCC, we recompute the GrF and GrM values at its two children nodes, in order to take into account that GrF paths from these subtrees could help lower the storage cost of c_1. Please note that the GrF*

Node	$GrM_i.b$	$GrM_i.sp$	$GrF_i.b$	$GrF_i.sp$	$State_i$	chM_i	chF_i
8	288	65	288	65	000	11	01
9	2	65	2	65	000	11	01
10	0	65	0	65	000	11	01
11	242	65	242	65	000	11	01
12	0	65	0	65	000	11	01
13	0	65	0	65	000	11	01
14	0	65	0	65	000	11	01
15	648	65	648	65	000	11	01
4	288	65	292	98	000	01	10
5	242	65	342	98	000	10	11
6	0	65	0	65	000	11	01
7	648	65	972	98	000	10	11
2	242	33	242	33	000	11	01
3	1134	99	1134	99	000	11	11
1	1134	99	0	0	110	10	00
0	1134	99	0	0	100	10	00

Table 4.7: *Computed values after marking the first selected HCC for storage*

Node	$GrM_i.b$	$GrM_i.sp$	$GrF_i.b$	$GrF_i.sp$	$State_i$	chM_i	chF_i
8	288	65	288	65	000	11	01
9	2	65	2	65	000	11	01
10	0	65	0	65	000	11	01
11	242	65	242	65	000	11	01
12	0	65	0	65	000	11	01
13	0	65	0	65	000	11	01
14	0	65	0	65	000	11	01
15	0	0	0	0	110	00	00
4	288	65	292	98	000	01	10
5	242	65	342	98	000	10	11
6	0	65	0	65	000	11	01
7	0	65	0	0	100	01	00
2	584	131	584	131	000	11	11
3	0	65	0	0	100	01	00
1	584	131	0	0	100	01	00
0	584	131	0	0	100	10	00

Table 4.8: *Computed values after marking the second selected HCC for storage*

values at nodes c_2 and c_3 both change (see Table 4.7), compared to the values in Table 4.6. Then, moving bottom-up we need to compute the GrF_1, GrM_1, GrF_0 and GrM_0 values, while properly setting the chM and chF bitmaps at nodes c_1 and c_0. The calculated entries at each node after marking for storage nodes c_0 and c_1 are depicted in Table 4.7.

In Table 4.8 we depict the calculated entries after the algorithm stores the next HCC, which contains the coefficients c_{15}, c_7 and c_3, and combines it with the first selected HCC. This can be easily identified by examining the State bitmaps. The 5 entries that are set at the first bit (from the left) of these bitmaps translate to 5 stored coefficient values. The 1 entry that is set at the second bit of these bitmaps translates to 1 different HCCs. Since c_{15} does not have any children nodes, we do not needs to recompute the GrF and GrM at any of its descendant nodes. However, since c_1 seizes to be the bottom-most coefficient at a HCC, the GrF_2 and GrM_2 values are recalculated to take into account that no path storing c_2 can lower the storage cost of c_1.

At this stage of the algorithm, the last HCC, containing nodes c_{11}, c_5 and c_2 can be stored.

4.5.3 Storing the Selected Solution

The process of storing the selected HCCs follows a preorder traversal of the nodes in the wavelet tree. At each visited node c_i, its input is a set (possibly empty) of *straddling* coefficient values. This set corresponds to coefficient values that belong to the same HCC, but where the lowest node in that HCC has not yet been visited. Any time the algorithms reaches a node c_i where the two bits $State_i(0)$ and $State_i(1)$ are both set, then the index/coordinate of c_i and its coefficient value along with the straddling coefficient values form a HCC. In this case, the input list to the both subtrees of c_i will be empty.

If only the $State_i(0)$ is set, but not the bit $State_i(1)$, then depending on the value of $State_i(2)$ the value c_i is attached to the list of straddling coefficient values for the appropriate subtree of c_i (the input list to the other subtree will be empty). If, finally, $State_i(0)$ is not set, then we simply recurse to the two subtrees with their inputs being empty lists of straddling coefficients.

4.5.4 Space and Running Time Complexities

For each node of the wavelet tree there are $O(1)$ stored variables. Thus, the needed space is $O(N)$. At the initialization step, the calculation of the GrM_i, GrF_i, chM_i and chF_i variables requires $O(1)$ time. Then, the algorithm repeatedly marks at least one coefficient for storage. Thus, at most $O(\frac{B}{S_2})$ steps can be performed. At each step a path originating at the root of the wavelet tree is traversed in order to mark for storage the nodes in GrM_0. This process visits at most $O(\log N)$ nodes. At each node, the recalculation of the GrM_i, GrF_i, chM_i and chF_i variables requires $O(1)$ time. Finally, the storage of the marked coefficients can be achieved in a single pass of the wavelet tree. Thus, the overall running time complexity is $O(N+\frac{B}{S_2}\log N) = O(N+B\log N)$. Note that the running time complexity are on par with that of constructing a conventional synopsis — hence, no significant increase in data processing time is expected (see also Section 4.8).

Theorem 4.5.1. *The* HCGreedyL2 *algorithm constructs a HCWS given a space budget of B, in $O(N + B\log N)$ time using $O(N)$ space.*

4.6 HCGreedyL2-Str: A Streaming Greedy Algorithm

In order for our algorithms to adapt to streaming environments, we propose a streaming greedy algorithm, termed as HCGreedyL2-Str in our discussion, for our optimization problem. As expected, the HCGreedyL2-Str algorithm shares some common characteristics with the HCGreedyL2 algorithm in the way that it constructs candidate HCCs for storage.

4.6.1 Data Structures

The algorithms proceeds by reading the data values one by one and by updating the (normalized) values of the wavelet coefficients. Note that the total number of data values to be read does not need to be known in advance, since the normalized value of a coefficient depends only on the number of data values that lie beneath it in the wavelet tree (and, thus, from the difference in levels between the node and

leaf coefficient values in the wavelet tree). This process has well been documented in prior work [44].

When reading the n-th data value, the values of the wavelet coefficients that lie in path(n) are updated. According to Definition 4.3.2, a wavelet coefficient is *closed* only when all the data values that beneath it in its wavelet tree have been read. Depending on the value of n, the number of coefficients that become closed due to a new data value ranges from 0 to $\log n + 1$. These newly closed coefficients all belong to the bottom portion of $path(n)$ that originates from the last read data value and proceeds upwards in the wavelet tree until $path(n)$ reaches the last wavelet tree node for which the data value belongs to its right subtree. Our HCGreedyL2-Str algorithm processes these newly closed nodes of the wavelet tree in a bottom-up fashion.

At each step of the algorithm, the current selection of HCCs is stored in a min-heap structure where the HCCs are ordered based on their per space benefit.[4] Each HCC is identified by its bottommost coefficient. We defer a detailed description and the implementation of this min-heap structure until later in this section.

The min-heap does not store each HCC explicitly, but rather a pointer to a structure containing: (i) The HCC; (ii) The benefit of the HCC; and (iii) The required space for the HCC. Please note that in order to guarantee that swapping any pair of HCCs in the min-heap can be performed in $O(1)$ time (and thus guarantee the worst time complexity of the First(), Pop() and Insert() operations), we cannot simply store the HCCs in the min-heap, due to their variable size. We finally note that the number of different HCCs stored in the min-heap is obviously $O(\frac{B}{S_2}) = O(B)$.

Another important characteristic of our HCGreedyL2-Str algorithm is that it does not fully combine the stored HCCs, even though it accurately estimates their space requirements. This means that there may exist pairs of HCCs (i.e., HCC h_A and HCC h_B) in the min-heap such that parent(top(h_A)) = bottom(h_B). In such a case, even though h_A and h_B are not combined in one HCC, the storage overhead for bottom(h_B) is correctly set to S_2 in our algorithm. We explain in Section 4.6.2 why our HCGreedyL2-Str algorithm utilizes such an approach of storing HCCs.

Besides the min-heap structure our HCGreedyL2-Str algorithm also utilizes two hash tables, termed as *TopCoeff* and *BottomCoeff*, with a maximum of $O(\frac{B}{S_2})$ entries each. The *TopCoeff* (*BottomCoeff*) hash table maps the coordinate c_i of a coefficient to the stored HCC h_A in the min-heap, such that c_i = top(h_A) (c_i = bottom(h_A)). If the coordinate c_i is not the top (bottom) coefficient value stored in any HCC, then the *TopCoeff* (*BottomCoeff*) hash table does not contain an entry for it.

4.6.2 Detailed Operations

Operations at each node. For each processed node c_i our HCGreedyL2-Str algorithm generates a straddling candidate HCC, termed as $SGrF_i$. This straddling HCC is similar to GrF_i, in that it corresponds to the non-stored candidate path in the node's subtree with the estimated maximum per space benefit when storing c_i. Thus, its computation is similar, with the only difference that due to the streaming nature of the algorithm and the bottom-up way of processing closed coefficients,

[4] We can alternatively use any data structure, such as an AVL-tree, which provides a worst case cost of $O(\log B)$ for the (i) search of the stored item with the minimum per space benefit; (ii) the insertion of an item; and (iii) the deletion of an item.

there is no way that c_i has already been stored in a HCC. Thus, the only choices considered for generating $SGrF_i$ are restricted to:

- Simply storing c_i. The space requirements of this choice is S_2 if either c_{2i} or c_{2i+1} has been stored, or S_1, otherwise. Please note that if c_{2i} or c_{2i+1} has been stored, then these coefficients must be the top coefficients in a stored HCC. This can be checked in $O(1)$ time by looking at the *TopCoeff* hash table. Let *Ben1* denote the per space benefit of this choice.

- Combining c_i with $SGrF_{2i}$ ($SGrF_{2i+1}$). The space requirements for $SGrF_i$ in this case is S_2+$SGrF_{2i}.sp$ (resp., S_2+ $SGrF_{2i+1}.sp$). Let *Ben2* (resp., *Ben3*) denote the per space benefit of this combination.

Given the aforementioned choices, $SGrF_i$ is set to:

1. $c_i \cup SGrF_{2i}$, if $Ben2 = \max\{Ben1, Ben2, Ben3\}$ and $Ben2$ is larger or equal to the per space benefit of $SGrF_{2i}$. In this case, $SGrF_{2i+1}$ cannot be of any further use in upper levels of the wavelet tree. Thus, it is checked for insertion to the min-heap, by comparing its per space benefit to that of the stored HCC with the minimum per space benefit.

2. $c_i \cup SGrF_{2i+1}$, if $Ben3 = \max\{Ben1, Ben2, Ben3\}$ and, further, $Ben3$ is larger or equal to the per space benefit of $SGrF_{2i+1}$. In this case, $SGrF_{2i}$ cannot be of any further use in upper levels of the wavelet tree. Thus, it is checked for insertion to the min-heap, by comparing its per space benefit to that of the stored HCC with the minimum per space benefit.

3. c_i, otherwise. In this case, $SGrF_{2i}$, $SGrF_{2i+1}$ are checked in succession for insertion to the min-heap, by comparing their per space benefit to that of the stored HCC with the minimum per space benefit.

Please note that, in the HCGreedyL2-Str algorithm, once we have computed the $SGrF_i$ coefficient for any node c_i, we no longer need to keep in main memory the straddling paths of its two subtrees.

Operations of the Min-Heap. We now present the basic operations of the Min-Heap structure.

1. First(): Returns the stored HCC with the minimum per space benefit. This is straightforward. The operation requires $O(1)$ time.

2. Pop(): Removes the First() item. The operation adjusts the size of the Min-Heap, based on two factors:

   - The size of the removed HCC, termed as h_A in our discussion. This is available in the third field of the item (see Section 4.6.1 on how HCCs are stored).
   - Whether removing this item requires adjusting the space of some other HCC h_B. This case occurs when parent(top(h_A)) = bottom(h_B) and the other child coefficient of bottom(h_B) is not currently stored in the Min-Heap. The former can be tested by first probing the *BottomCoeff* hash table to see if parent(top(h_A)) exists as the bottom-most coefficient in a

procedure Insert(h_A)
Input: HCC h_A to insert into the Min-Heap.
1. A min-heap structure *hcs* is used to maintain the currently selected HCCs for storage
2. Each entry in *hcs* has 3 fields: (1) *hc:* the stored HCC, (2) *ben:* benefit of the HCC, (3) *sp:* space needed for storing the HCC.
3. $UsedB \leq B$ denotes the true space required to compactly store the HCCs of the Min-Heap.
4. tophc = hcs.First()
5. lastPopped = $\emptyset$
6. **while** $UsedB + h_A.sp > B$ AND $\frac{tophc.ben}{tophc.sp} < \frac{h_A.sp}{h_A.ben}$ **do**
7. lastPopped = tophc
8. hcs.Pop(). Also update TopCoeff and BottomCoeff hash tables
9. Update $UsedB$ based on discussion in Section 4.6.2
10. **endwhile**
11. Insert h_A in the heap using standard heap operation. Update TopCoeff, BottomCoeff and $UsedB$.
12. **if** $UsedB < B$ **then** Trim sufficient coefficient values from lastPopped and reinsert it in the Max-Heap.

end

Figure 4.2: *Sketch of Insert algorithm*

stored HCC. The latter can be tested by then probing the *TopCoeff* hash table for the other child of bottom(h_B). If both conditions are satisfied, then the space requirements of h_B are adjusted and the standard heap procedure *heapifyUp()* is invoked in order to make sure that no conditions are violated in the path of the heap between the updated node and the root of the heap. This *heapifyUp()* operation requires $O(\log B)$ time.

Thus. the Pop() operation requires a total of $O(\log B)$ time.

3. Insert(h_A): Inserts the given HCC h_A in the Min-Heap. This operation is presented in Algorithm 4.2. The running time requirements of the Insert() operation depend on the size of the inserted HCC and the number of popped HCCs (Lines 6-10). In the worst case, for a HCC containing $O(\log n)$ coefficient values, the operation may require $O(\log n \cdot \log B)$ time. However, an interesting observation is that for any HCC containing more than one coefficient values, the insert operation is performed only for the top coefficient value of the HCC. Thus, the amortized cost of the insert operation per processed wavelet coefficient remains $O(\log B)$.

4. Parse(): Scans the min-heap and extracts the stored HCCs in a compact form with size at most B. In order to perform this step we need to combine the HCCs stored in the Min-Heap. When checking each stored HCC h_A, we also check to see if there exists another unprocessed HCC h_B that needs to be processed before h_A, and such that h_A can be attached on top of h_B (so that their bitmaps are combined). This requires checking the *TopCoeff* hash table for the two children of bottom(h_A). This step essentially creates a recursive processing of the HCCs similarly to a topological sort. Since the min-heap

cannot store more than $O(\frac{B}{S_2})$ entries, this operation requires a total of $O(B)$ time.

A question that naturally arises is why we chose to store the current selection of the HCCs in a way that does not aggressively combine them, even though storage dependencies are indeed exploited. If we had pursued to aggressively merge stored HCCs, coefficient values with large benefits might end up in HCCs with several other small coefficient values, e.g., a HCC containing the coefficient values $\langle 800, 10, 20, 5\rangle$. This could potentially lead to HCCs with small to medium overall per space benefit, even though a part of them exhibits a large per space benefit. Please note that in the HCGreedyL2 algorithm, such a problem did not exist, as HCCs were attached to existing HCCs after exhibiting globally the best estimated per space benefit. Due to the streaming nature of the HCGreedyL2-Str algorithm, this global estimate cannot be achieved since future parts of the wavelet tree have not been unveiled yet. Thus, we need to be careful in our decisions to aggressively merge HCCs.

4.6.3 Space and Running Time Complexities

Based on the analysis presented in Section 4.6.2, the operations associated with inserting a HCC in the Min-Heap cost a total of $O(\log B)$ time. The insert operation at some nodes may exhibit a higher cost but, as we explained in Section 4.6.2, this cost is amortized over the coefficient values that comprise the HCC. The space requirements are those of the Min-Heap, the two hash tables and the straddling coefficients. The Min-Heap and each hash table requires $O(B)$ space. Parsing the Min-Heap to extract the synopsis also requires $O(B)$ time. There can be at most $O(\log n)$ straddling coefficients, of total size $O(\log^2 n)$. Thus, the amortized running time requirements per processed data item are $O(\log B)$, while the space requirements are $O(B + \log^2 n)$.

4.7 Extensions and Remarks

4.7.1 Multiple Dimensions

The Haar decomposition of a D-dimensional data array A results in a D-dimensional wavelet-coefficient array W_A with the same dimension ranges and number of entries. (The full details as well as efficient decomposition algorithms can be found in [13, 88].) Consider a D-dimensional wavelet coefficient W in the wavelet-coefficient array W_A. W contributes to the reconstruction of a D-dimensional rectangular region of cells in the data array A (i.e., W's *support region*). Further, the sign of W's contribution ($+W$ or $-W$) can vary along the quadrants of its support region. The blank areas for each coefficient correspond to regions of A whose reconstruction is independent of the coefficient, i.e., the coefficient's contribution is 0. Each data cell in A can be accurately reconstructed by adding up the contributions (with the appropriate signs) of those coefficients whose support regions include the cell.

Wavelet tree structures for multidimensional Haar wavelets can be constructed (in linear time) in a manner similar to those for the one-dimensional case, but their semantics and structure are somewhat more complex. A major difference is that, in a D-dimensional wavelet tree, each node (except for the root, i.e., the overall average)

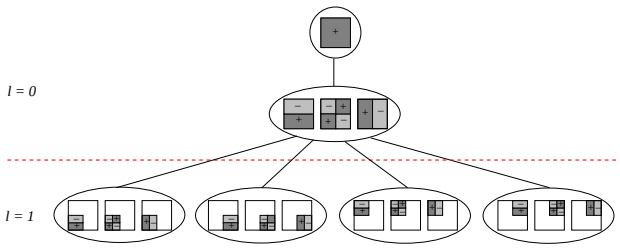


Figure 4.3: *Wavelet tree structure for the sixteen two-dimensional Haar coefficients for a* 4×4 *data array (data values omitted for clarity)*

actually corresponds to a *set* of $2^D - 1$ wavelet coefficients that have the same support region but different quadrant signs and magnitudes for their contribution. Furthermore, each (non-root) node t in a D-dimensional wavelet tree has 2^D children corresponding to the quadrants of the (common) support region of all coefficients in t.[5] If the maximum domain size amongst all dimensions is N_{max}, the height of the wavelet tree will be equal to $\log N_{max}$. Note that the total domain size N can be as high as $N = N_{max}^D$ when all dimensions have equal domain size. Figure 4.3 depicts an example wavelet tree structure for a two-dimensional 4×4 dataset.

A *multidimensional hierarchically compressed wavelet synopsis* (MHCWS) groups nodes (not coefficients) into paths and thus requires additional information as to which coefficients of each node are included in the synopsis.

Definition 4.7.1. *The composite value NV of some node in the multidimensional wavelet tree is a pair* $\langle \text{NVBIT}, V \rangle$ *consisting of:*

- *A bitmap* NVBIT *of size* $2^D - 1$ *identifying which coefficient values are stored. The number of stored coefficient values is equal to the bits of* NVBIT *that are set.*
- *The set V of stored coefficient values.*

Having properly defined the composite value of a node we can now define a multidimensional hierarchically compressed wavelet coefficient as follows.

Definition 4.7.2. *A multidimensional hierarchically compressed (MHCC) wavelet coefficient is a triplet* $\langle \text{BIT}, C, \mathcal{NV} \rangle$ *consisting of:*

- *A bitmap* BIT *of size* $|\text{BIT}| \geq 1$*, denoting the storage of exactly* $|\text{BIT}|$ *node values.*
- *The coordinate/index C of any stored coefficient in the bottommost stored node.*
- *A set* $\mathcal{NV}$ *of* $|\text{BIT}|$ *stored composite values.*

We must note here that at any MHCC the coordinate of any stored coefficient in its bottommost stored node can be used, since the bitmap of that node's composite value can help determine which other coefficient values from the same node have also been stored.

[5]The number of children (coefficients) for an internal wavelet tree node can actually be less than 2^D (respectively, $2^D - 1$) when the sizes of the data dimensions are not all equal. In these situations, the exponent for 2 is determined by the number of dimensions that are *"active"* at the current level of the decomposition (i.e., those dimensions that are still being recursively split by averaging/differencing).

We now describe the necessary changes to the HCDynL2 and HCGreedyL2 algorithms for multidimensional datasets. The modifications to HCApprL2 are similar to the ones of HCDynL2.

Changes to HCDynL2. The extensions to the HCDynL2 algorithm are analogous to the corresponding extensions of prior DP techniques [33] to multi-dimensional datasets. In particular, when obtaining an optimal MHCWS given a space budget B, the algorithm given budget B should consider (i) the optimal benefit $\mathrm{M}[i, B]$ assigning space B to the subtree rooted at node i; and (ii) the optimal benefit $\mathrm{F}[i, B]$ assigning space B to the subtree rooted at node i *when at least one of the coefficients* of node i is forced to be stored (i.e., a composite value of the node is stored). The principle of optimality also holds in this case for $\mathrm{M}[i, B]$ and $\mathrm{F}[i, B]$, implying that optimal benefits at a node can be computed from optimal solutions of the node's subtrees.

At each node of the wavelet tree, the optimal algorithm needs to decide how many coefficients, if any, of this node should be stored, whether they should be attached to some path of its children subtrees, and how much space to allocate to each child subtree. It should be noted that we only need to decide how many coefficients (from 1 to $2^D - 1$) of each node should be stored, as it can be easily shown that among all coefficient sets of k values, the set containing the coefficients with the k highest absolute normalized values exhibits the best benefit.

When the algorithm checks if a node should be included in the optimal solution but cannot be attached to any path of the children subtrees, the space requirement for this node is a function of the number $k \leq 2^D - 1$ of coefficients to be included (a choice to be made): $S_1(k) = \mathrm{sizeof}(Coords) + 2^D + k \cdot \mathrm{sizeof}(Value)$. Similarly, when the node at question can be attached to some path the space requirement is again a function of the number k of selected coefficients: $S_2(k) = 2^D + k \cdot \mathrm{sizeof}(Value)$. Note that only in the first case the node "pays" for the overhead $\mathrm{sizeof}(Coords)$ of creating a new $MHCC$.

At each node of the wavelet tree the algorithm must perform two tasks: (i) sort the $2^D - 1$ coefficients of this node in $O(D2^D)$ time and $O(2^D)$ space; and (ii) for each space budget $0 \leq b \leq B^*$ choose the optimal split of space among the coefficients of this node and the 2^D children nodes. Note that, because a subtree rooted at a node at height l of the wavelet tree can have up to $O(2^{Dl})$ nodes, the maximum allotted space at such a node is $B^* = \min\{B, O(2^{Dl})\}$. The second task can be performed in $O(2^D B^{*2})$, by solving a dynamic programming recurrence on a binary tree of height D constructed over the children nodes — for details refer to [33]. Using similar analysis with Section 4.3 and since there are at most $\frac{N_{max}^D}{2^{Dl}} = \frac{N}{2^{Dl}}$ nodes at height l it follows that the space complexity becomes $O(2^D N \log B)$, whereas the time complexity becomes $O(2^D NB)$.

Finally, note that the ratio of benefits between HCDynL2 and the traditional technique can become as high as $\frac{1+\log N_{max} \times (2^D-1)}{m}$ for $m = \lfloor \frac{S_1 + \log N_{max} \times (2^D-1) \times S_2}{S_1 - 1} \rfloor$. The increased maximum value of the above ratio, when compared to the one-dimensional case, is not surprising, as in multidimensional datasets the existence of multiple coefficient values within each node of the wavelet tree provides far more opportunities to exploit hierarchical relationships amongst stored coefficients, in order to reduce the storage overhead of their coordinates. Also, note that in the multidimensional case this storage overhead (and thus the size of S_1) increases with the number of

dimensions, due to the increase in the number of the coefficient coordinates.

Changes to HCGreedyL2. For the HCGreedyL2 algorithm, when considering whether to include a node in a MHCC, or to attach it to a MHCC originating from one of the node's subtrees, we utilize the node's composite value that results in the best per space benefit. This can be accomplished by (i) sorting the node's coefficient values based on their normalized value; (ii) for $1 \leq j \leq 2^D - 1$ computing the per space benefit of the composite value that stores the node's j largest normalized values; and (iii) selecting the composite value with the overall best per space benefit. For nodes where, at some point of the algorithm's execution, some coefficient values have already been selected for storage, we only need to consider in the above case coefficient values that have not already been included in the solution and properly determine the space needed for their storage. The HCGreedyL2 algorithm, given a budget of B, requires $O(2^D N)$ space and only $O(D2^D N + 2^D B \log N_{max})$ time.

4.7.2 Dealing with Massive Datasets

In order to improve the running time and space requirements of our algorithms for massive datasets, we can employ an initial thresholding step to discard coefficients with small values and apply our algorithms to the remaining $N_z \ll N$ coefficients. Such an approach is commonly followed for constructing wavelet synopses; the work in [88], for example, maintains only N_z coefficients after the decomposition to deal with sparse datasets of $N_z \ll N$ tuples. Preserving only N_z coefficients means that there can be at most N_z "important" nodes in the wavelet tree (in practice much fewer, as many large coefficients usually reside in a single node), which is a significant decrease compared to $N/2^D$, the total number of nodes.

More precisely, it is easy to see that all of our algorithms need to perform some computations to nodes that either (i) contain a non-zero coefficient value; or (ii) contain non-zero coefficient values at (at least) two of their subtrees. Thus, the total number of nodes where some computation needs to be performed is $O(2N_z - 1) = O(N_z)$. By sorting these nodes using a pre-order traversal it is easy to mark for each node: (i) the closest ancestor $anc(i)$ of i where computation needs to be performed; (ii) the subtree of $anc(i)$ that follows i; and (iii) the first descendant of i where computation needs to be performed. This process requires $O(N_z \log N_z)$ time, but allows for the execution of the algorithms with complexities that depend on N_z rather than N. Of course, some care is needed because the children of each node in the above "sparse" wavelet tree are not direct descendants, thus requiring proper calculation of the space needed when storing a node's composite value and combining it with a MHCC originating from one of the node's subtrees. Thus, when attaching a composite value to a MHCC that lies j levels below it in the sparse wavelet tree, the value of S_2 must be set as follows: $S_2(k) = j \times (2^D - 1) + j + k \cdot \text{sizeof}(Value)$. The first summand in the above formula is due to the storage of the NVBit bitmaps for both the current node and all the intermediate, missing nodes until reaching the MHCC of the descendant node. The second summand determines the number of these bitmaps, while the third summand is due to the storage of k coefficient values in the node. Please note that each node of the sparse wavelet tree may exhibit different S_2 values for each of its subtrees, due to the potentially different resolution levels of each subtree's root node.

4.7.3 Optimizing for Other Error Metrics

All algorithms presented here can be made to optimize for any weighted $\mathcal{L}_2^w$ error metric. These error metrics include the sum squared relative error with sanity bound s (set $w_i = \frac{1}{\max\{d_i,s\}}$), and the expected sum squared error when queries are drawn from a workload distribution, in which case the weights correspond to the probability of occurrence for each query (set $w_i = p_i$).

For the weighted $\mathcal{L}_2^w$ metric and using the standard Haar decomposition process the Parseval theorem does not apply and hence Problem 4.2 does not follow from Problem 4.1. However the recent work of [62] demonstrated that the Parseval theorem applies when the decomposition process is altered to incorporate the weights. The result is a modified Haar basis for which the Parseval applies and, therefore, an analogous to Problem 4.2 formulation exists and our algorithms require no additional changes.

4.7.4 Query Performance Issues

For a synopsis size of B, due to the use of a variable-length header for the stored HCC coefficients, the retrieval of a single coefficient value requires $O(B)$ time, in contrast to $O(\min\{B, \log N\})$ time for the conventional wavelet synopses, where binary search is employed if the stored coefficients are sorted based on their coordinates. While this may seem as a potentially large increase in the resulting query time, we need to make two important observations: (i) The used synopses are typically memory resident and of small size ($B \ll N$); and (ii) To answer even point queries, $O(\log N)$ coefficients need to be retrieved. The number of retrieved coefficients is increased even more if a query that requires the evaluation of multiple individual data values (or data values in multiple areas of the data) is issued. This has the effect that a linear scan of the synopsis, to retrieve at batch all the desired coefficients, even in conventional wavelet synopses, is often as efficient as performing a logarithmic (or larger) number of binary searches in the synopsis. Thus, we expect that any potential running time deterioration due to the use of our proposed technique will be minimal. On the other hand, the improvements in the obtained accuracy achieved by the use of HCWS can be significant, as shown in Section 4.8.

4.8 Experiments

In this section, we present an extensive experimental study of our proposed algorithms for constructing hierarchically compressed wavelet synopses over large datasets. Our objective is to evaluate the scalability and the obtained accuracy of our algorithms when compared to conventional synopses. Our main findings include:

• **Improved Space Utilization.** The algorithms presented in this work create HCWS that consistently exhibit significant reductions in terms of the sum squared error of the approximation due to the improved storage utilization of the selected wavelet coefficients.

• **Efficient, Near-Optimal Greedy HCWS Construction.** Even though the HCGreedyL2 algorithm does not provide any guarantees on the quality of the obtained solution, in all of our experiments it provided near optimal results. At the same

time, the HCGreedyL2 algorithm exhibits running time and space requirements on par with the conventional synopsis construction method. Moreover, our proposed HCGreedyL2-Str algorithm consistently produces HCWS with errors very close to those of the HCGreedyL2 algorithm.

Techniques and Implementation Details. We compare the algorithms HCDynL2, HCApprL2, HCGreedyL2, HCGreedyL2-Str introduced in this chapter against the conventional synopsis construction algorithm denoted as Classic. The Classic algorithm utilizes a heap to identify the coefficients with the largest absolute normalized values, while not exceeding the available space budget. All algorithms were implemented in C++ and the experiments reported here were performed on a 2.4 GHz machine.

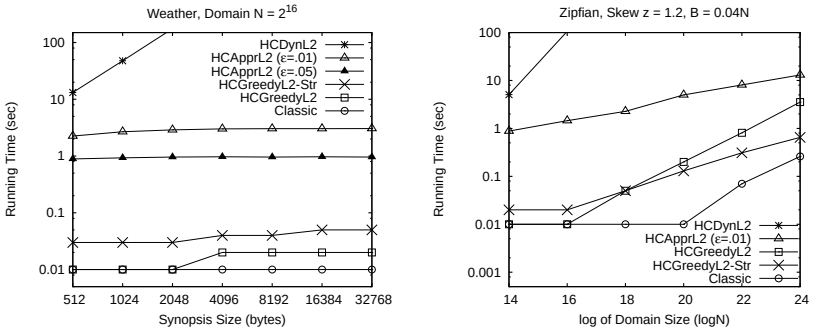
Datasets. We have performed an extensive experimental study with several one-dimensional synthetic and real-life datasets; we present here the most significant findings. Each synthetic dataset, termed `Zipfian`, is produced by generating 50 different Zipfian distributions with the same skew parameter (where the values are placed in random locations of the data) and then summing up these 50 smaller datasets. We vary the domain size from $N = 2^{14}$ up to $2^{24} = 16,777,216$ and examine two values of the Zipfian parameter, $z = 0.7$ and $z = 1.2$, i.e., average and high skew respectively. The first real dataset, denoted as `Weather`[6], contains $N = 65,536$ solar irradiance measurements obtained from a station at the University of Washington. The second real dataset, denoted as `Light`, consists of light measurements from the Intel Labs dataset [24]. In all experiments involving `Light`, we use the measurements of the sixth mote (sensor) of this dataset.

Performance Metrics. We first investigate the running time scalability of our algorithms when varying the available synopsis budget, the data domain size and the ϵ parameter for the HCApprL2 algorithm. In order to assess the quality of the constructed HCWS we measure the sum squared error (SSE). To emphasize on the effectiveness over conventional synopses: (i) we explicitly measure the SSE increase of Classic relative to HCGreedyL2; and (ii) show how much more space (*space savings*) we would need to allocate to a conventional synopsis in order for it to become as accurate as our constructed HCWS. In a graph depicting the resulting SSE by all algorithms when varying the synopsis size, the SSE increase in absolute value can be measured at each point by the vertical distance between the graph of the Classic technique from the graph of either the HCDynL2, the HCApprL2, the HCGreedyL2 or the HCGreedyL2-Str algorithm. Correspondingly, in the same graph, the *space savings* of our algorithms can be (roughly) measured, for any space budget assigned to our algorithms, by the horizontal distance to the right, starting of course at the point of the graph corresponding to our technique and for the desired space budget, until we meet the graph (error) of the Classic algorithm. Recall that the goal of deploying a HCWS is to achieve better storage utilization and to improve the accuracy of the synopsis by storing, within a given space budget, a larger number of "important" coefficient values than a traditional wavelet synopsis. The space savings essentially provide us with an insight on how many "important" wavelet coefficients the HCWS contains, in addition to the ones selected by the Classic algorithm, that are responsible for the achieved SSE reduction (and, thus, how much can our algorithms exploit hierarchical relationships amongst coefficient values selected for storage). The com-

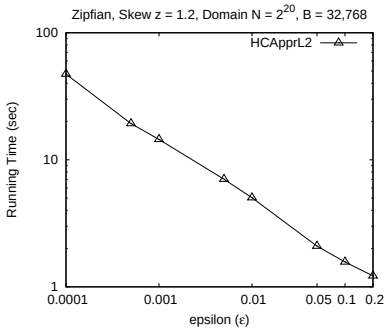
[6]Data available at: `http://www-k12.atmos.washington.edu/k12/grayskies/`

bination of the two performance metrics also reveals some helpful characteristics on the distribution of the coefficient values. For example, assume that our algorithms consistently result in half the error achieved by the Classic algorithm, but that the space savings increase (decrease) as the synopsis size increases. This implies that as the synopsis size increases, and more coefficient values are stored, the number of non-stored coefficient values that are responsible for half of the *remaining* SSE also increases (decreases), since the Classic algorithm requires increasingly more (less) space to reduce its SSE by 50%.

Further, we explicitly measure the deviation of the error exhibited by the solution of our HCGreedyL2 algorithm, when compared to the corresponding optimal error exhibited by the solution of our HCDynL2 algorithm, when varying either the available synopsis budget, or the data domain size. We also measure the errors achieved by our HCGreedyL2-Str algorithm, when compared to the corresponding errors of our HCGreedyL2 algorithm. Finally, we plot the approximation ratio achieved by the HCApprL2 algorithm against the theoretical bound.



(a) Running Time vs Synopsis Size (b) Running Time vs Domain Size

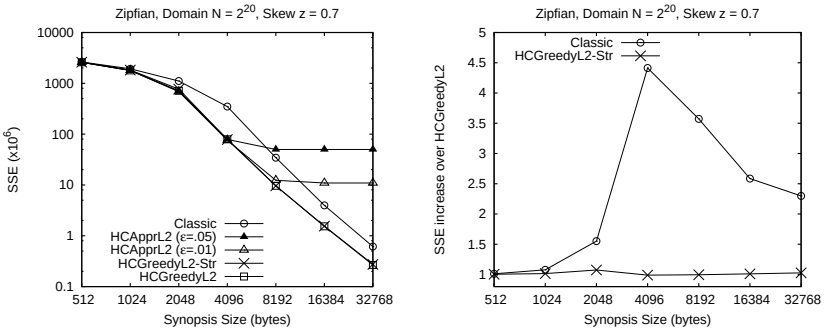


(c) Running Time vs ϵ

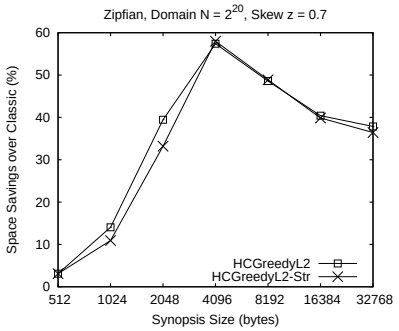
Figure 4.4: *Running time performance of all algorithms*

In the following we present the experimental results.

Scalability. Figure 4.4 investigates the scalability, in terms of the total running time, for all methods while the synopsis size and the domain size is varied. For the HCApprL2 algorithm we also plot its running time when varying the approximation parameter. Figure 4.4(a) presents the running time for the `Weather` dataset when the available synopsis size increases from 512 to 32,768 bytes. The approximation parameter for the HCApprL2 algorithm was set to $\epsilon = 0.05$ and 0.01. Please note that logarithmic axes are used for both the resulting running time and the synopsis size. In this experiment, the HCGreedyL2 and HCGreedyL2-Str algorithms consistently construct a HCWS within a few hundredths of a second, and almost as fast (with



(a) SSE vs Synopsis Size (b) SSE Increase vs Synopsis Size



(c) Space Savings vs Synopsis Size

Figure 4.5: *HCWS quality vs synopsis size for* `Zipfian`*,* $z = 0.7$*,* $N = 2^{20}$

an increase in running time by a factor between 2 and 5) as Classic constructs a conventional synopsis. The HCDynL2 algorithm could not construct large HCWSs within a reasonable time, as depicted on Figure 4.4(a), due to its linear dependency on B. Similar trends were observed for all datasets and, thus, the graphs for the HCDynL2 algorithm are often omitted.

Figure 4.4(b) illustrates the scalability of the algorithms as the domain size increases from 2^{14} up to 2^{24} for the `Zipfian` dataset with a skew parameter of 1.2. The synopsis size is set to a fixed percentage (4%) of the original data size. Therefore, the time complexity of HCDynL2 essentially becomes quadratic on the domain size. This is depicted on Figure 4.4(b), as the running time of HCDynL2 for domains larger than 2^{16} becomes prohibitive, while HCGreedyL2 can construct a HCWS in about 3.5 seconds, even for a domain size of 2^{24}. The running time of the streaming variant HCGreedyL2-Str increases at a lower rate than that of HCGreedyL2, as the domain size increases. This is attributed to the fact that the running time complexity for the HCGreedyL2-Str algorithm is based on a pessimistic case where every HCC tested for insertion in the min-heap requires $O(\log B)$ time. In practice, most of the HCCs in large domains do not have a sufficiently large per space benefit to be inserted into the min-heap, thus requiring only $O(1)$ time for them. Finally, note that even if it exhibits running times that are up to 2 orders of magnitude larger than the ones of HCGreedyL2, the HCApprL2 algorithm scales significantly better than the HCDynL2 algorithm.

Figure 4.4(c) plots the running time of HCApprL2 as the approximation parameter ranges from $\epsilon = 0.0001$ to 0.2 for the `Zipfian` dataset with a skew parameter of 1.2, $N = 2^{20}$ data values and a fixed value of $B = 32768$. As the approximation requirements relax, the running time of HCApprL2 decreases exponentially.

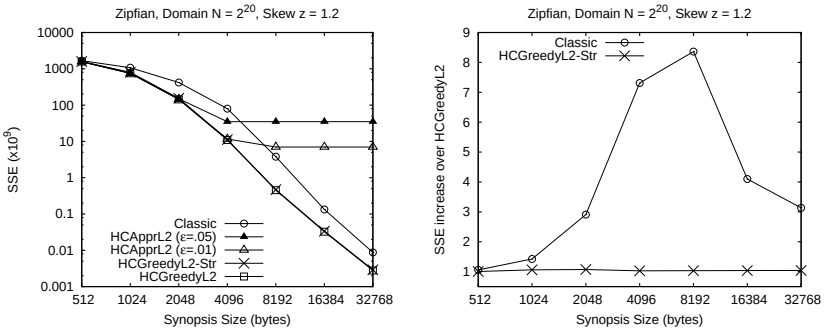
HCWS Quality. In Figures 4.5, 4.6, 4.7 and 4.8 we investigate the quality of the HCWS synopses for the four datasets, as we vary the synopsis size from 512 to 32,768 bytes. For all datasets, we measure the SSE of the resulting synopses.

Figure 4.5(a) plots the SSE for all methods on the `Zipfian` dataset with the average skew value. The HCGreedyL2 algorithm consistently constructs a synopsis with significantly smaller errors compared to a conventional synopsis. Moreover, the HCGreedyL2-Str algorithm achieves similar benefits, as its performance closely matches that of HCGreedyL2. On the other hand, the accuracy of the HCApprL2 algorithm quickly approaches the point where the algorithm manages to construct a synopsis that has captured a sufficiently large fraction $1/(1+\epsilon)$ of the data's energy (and it is, thus, certainly also within the same $1/(1+\epsilon)$ factor from the optimal algorithm) — hence, further increasing the budget leads to the HCApprL2 algorithm constructing the same synopsis. Figure 4.5(b) plots the *SSE increase* (i.e., the ratio of the SSE errors) of Classic and HCGreedyL2-Str over HCGreedyL2. We first observe that for a space budget of $B = 4096$, HCGreedyL2 constructs an HCWS that has almost 4.5 times less SSE than a conventional synopsis. HCGreedyL2-Str constructs synopses with similar SSE compared to HCGreedyL2. Comparing the two greedy heuristics, HCGreedyL2-Str achieves 2% lower SSE in the best case ($B = 4096$), and 7.4% larger SSE in the worst case ($B = 2048$), than HCGreedyL2. Figure 4.5(b) illustrates the space savings of the two greedy algorithms compared to a conventional synopsis that would achieve the same SSE. As the synopsis size increases, the space savings of our algorithms in absolute values (i.e., in bytes) increase as well. In relative terms (i.e., as a percentage to the synopsis size), the best case for our methods appears for $B = 4096$, where a HCWS requires 57.4% less space than a conventional synopsis. The space savings of HCGreedyL2-Str show a similar trend with a maximum savings of 58% for $B = 4096$.

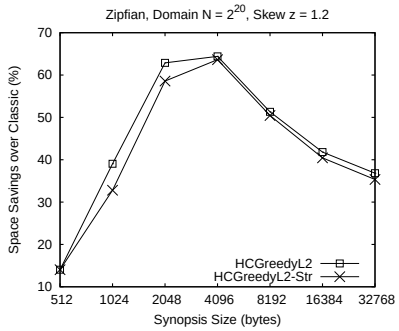
Figure 4.6 repeats the above setup using the `Zipfian` dataset with high skew ($z = 1.2$). The higher skew results in a more compressible dataset with the SSE decreasing rapidly with B, as depicted on Figure 4.6(a). In this dataset, constructing hierarchically compressed synopses proves highly beneficial as shown in Figures 4.6(b) and 4.6(c). HCGreedyL2 construct a synopsis with up to 8.3 times lower SSE than Classic (for $B = 8192$). Furthermore, the space savings of the HCGreedyL2 algorithm are significant (up to 64% for a synopsis size of $B = 4096$). Note that HCGreedyL2-Str constructs synopses with marginally increased SSE compared to HCGreedyL2 (up to 7% increase, with an average increase of 2%).

Figures 4.7 and 4.8 repeat the previous experimental setup for the real-life datasets, `Weather` and `Light`, respectively. For both datasets, the benefits, in terms of the reduction in the SSE, increase with the synopsis size. For the `Weather` dataset, the HCGreedyL2 algorithm results in up to 2.36 times lower SSE (for $B = 32768$), as shown in Figure 4.7(b). On the other hand, Figure 4.8(b) shows that in the `Light` dataset, the HCGreedyL2 algorithm achieves a reduction in SSE of up to 4.7 times (for $B = 32768$). For both real datasets, and for synopsis sizes larger than 1024 bytes, the space savings of our methods are consistently high (please note our earlier discussion that the benefits in absolute terms continuously increase in these cases as well, even though the relative space savings start decreasing at some point), as shown in Figures 4.7(c) and 4.8(c).

The effect of the domain size in the performance of our algorithms is illustrated in Figure 4.9. In this setup we use the `Zipfian` dataset with the high skew value

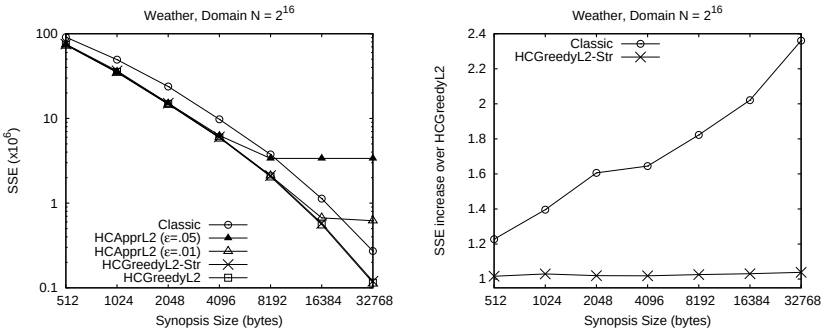


(a) SSE vs Synopsis Size (b) SSE Increase vs Synopsis Size

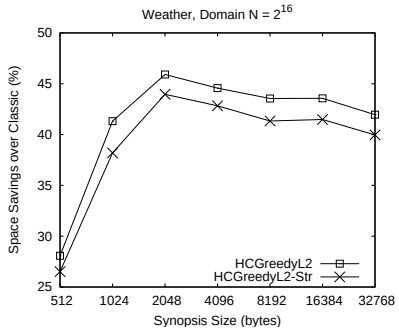


(c) Space Savings vs Synopsis Size

Figure 4.6: *HCWS quality vs synopsis size for* `Zipfian`*,* $z = 1.2$*,* $N = 2^{20}$



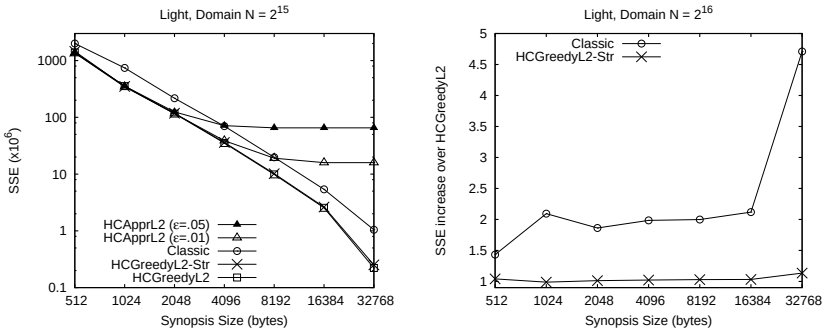
(a) SSE vs Synopsis Size (b) SSE Increase vs Synopsis Size



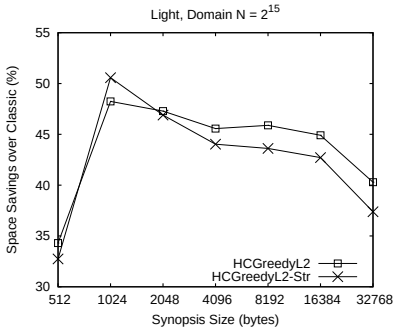
(c) Space Savings vs Synopsis Size

Figure 4.7: *HCWS quality vs synopsis size for* `Weather`*,* $N = 2^{16}$

($z = 1.2$) and vary the domain size from $N = 2^{14}$ up 2^{24}, while maintaining the synopsis size to 4% of N. Similar findings hold for other space ratios as well as for the average skew dataset. As seen in Figure 4.9(a), both greedy variants consistently



(a) SSE vs Synopsis Size (b) SSE Increase vs Synopsis Size



(c) Space Savings vs Synopsis Size

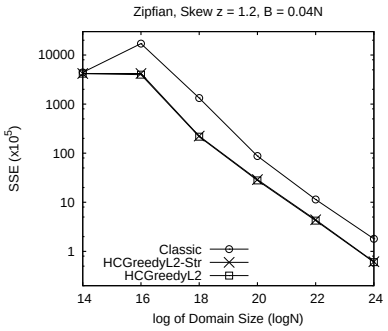
Figure 4.8: *HCWS quality vs synopsis size for* `Light`*,* $N = 2^{15}$

construct synopses with lower SSE (up to 7.4 times) than Classic. Similarly, our greedy heuristics are able to achieve significant space savings (up to 69% for the HCGreedyL2 algorithm and up to 66% for the HCGreedyL2-Str algorithm), compared to the Classic algorithm.

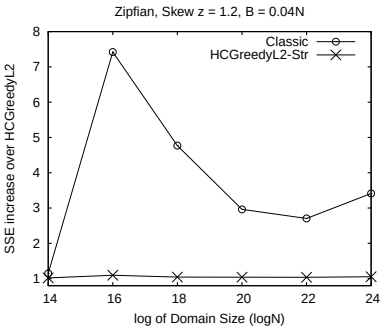
HCGreedyL2, HCGreedyL2-Str and HCApprL2 Accuracy. The HCGreedyL2 and HCGreedyL2-Str algorithms, as we have seen, require only frugal time and space in order to construct a wavelet synopsis when compared to the optimal HCDynL2 algorithm. A question that naturally arises is how close is the error of a HCWS constructed by the greedy algorithms to the one of the optimal HCWS. Thus, in the following set of experiments we measure the SSE increase incurred by HCGreedyL2 and HCGreedyL2-Str when constructing a HCWS — this is, essentially, the ratio between the errors of the greedy variants and the HCDynL2 algorithms.

Figure 4.10(a) shows the SSE increase ratio for the `Weather` dataset as the space budget is varied from 512 to 4096 bytes. It is easy to see that the error of the HCWS obtained by HCGreedyL2 (HCGreedyL2-Str) is always within 1.6% (4.6%) of the error achieved by the optimal HCWS. Figure 4.10(b) shows the SSE increase for the `Zipfian` dataset as the domain size varies from 2^{10} to 2^{15}, while the synopsis size is set to 1% of the original data. Such a setup is chosen so that the HCDynL2 algorithm, which provides the optimal HCWS, can execute within the available memory and within a time window of one hour. Again, the error of the HCWS obtained by HCGreedyL2 is within 2.2% of the error achieved by the optimal HCWS, while in 3 cases the HCGreedyL2 algorithm produced the optimal solution. Regarding the accuracy of HCGreedyL2-Str, note that in the worst case it produces HCWS with error which is within 12% (and with an average value of 4%) of the optimal.

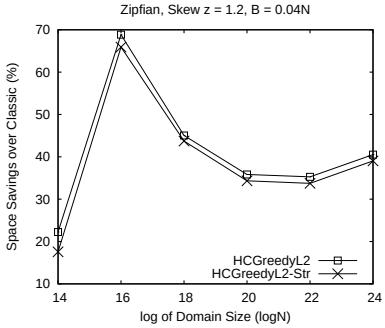
To measure the quality of HCApprL2, we plot the approximation ratio (benefit of



(a) SSE vs Domain Size

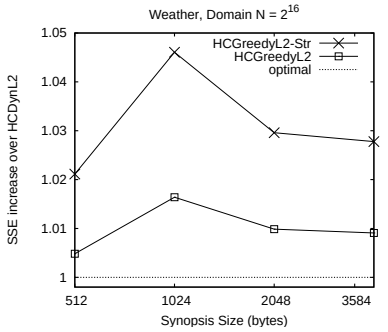


(b) SSE Increase vs Domain Size

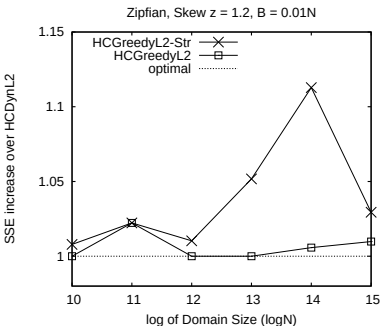


(c) Space Savings vs Domain Size

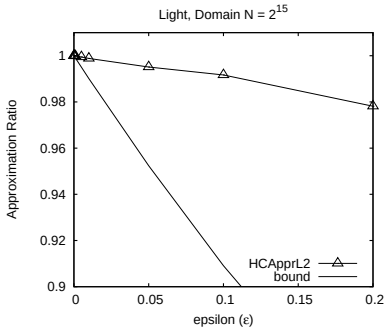
Figure 4.9: *HCWS quality vs domain size for* `Zipfian`*,* $z = 1.2$*,* $B = 0.04N$



(a) HCGreedyL2 Accuracy vs Synopsis Size



(b) HCGreedyL2 Accuracy vs Domain Size



(c) HCApprL2 Accuracy vs ϵ

Figure 4.10: HCGreedyL2*,* HCGreedyL2-Str*, and* HCApprL2 *accuracy*

constructed HCWS over the benefit of the optimal HCWS) for HCApprL2 as ϵ varies in Figure 4.10(c). Further, we also plot the theoretical bound of $\frac{1}{1+\epsilon}$ for reference. Observe that HCApprL2 consistently achieves a HCWS with approximation ratio significantly larger than the theoretical bound.

4.9 Summary

This chapter proposed a novel compression scheme for indexing wavelet synopses, termed Hierarchically Compressed Wavelet Synopses (HCWS). Our scheme seeks to improve the storage utilization of the wavelet coefficients and, thus, achieve improved accuracy to user queries by reducing the storage overhead of their coordinates. To accomplish this goal, our techniques exploit the hierarchical dependencies among wavelet coefficients that often arise in real datasets due to the existence of large spikes among neighboring data values and, more importantly, incorporate this goal in the synopsis construction process. We initially presented a dynamic programming algorithm, along with a streaming version of this algorithm, for constructing an optimal HCWS that minimizes the sum squared error given a space budget. We demonstrated that while in the worst case the benefit of our DP solution is only equal to the benefit of the conventional thresholding approach, it can often be significantly larger, thus achieving significantly reduced errors in the data reconstruction. We then presented an approximation algorithm with tunable guarantees leveraging a trade-off between synopsis accuracy and running time. Finally, we presented a fast greedy algorithm, along with a streaming version of this algorithm. We demonstrated that both of our greedy heuristics always exhibited near-optimal results in our experimental evaluation, with a running time on par with conventional thresholding algorithms. Extensions for multidimensional datasets, running time improvements for massive datasets and generalization to other error metrics were also introduced. Extensive experimental results demonstrate the effectiveness of HCWS against conventional synopsis techniques.

Chapter 5

Moving Objects Synopses

In this chapter we consider an environment of numerous moving objects, equipped with location-sensing devices and capable of communicating with a central coordinator. In this context we aim at constructing *moving object synopses* that summarize the important traits of the objects' movement.

We begin our investigation considering only the spatial dimensions of the moving objects data streams. We construct *spatial k-medoid synopses*; given a set of moving objects P, we are asked to choose k representative objects in P as the *medoids*. The optimal medoid synopsis minimizes the average Euclidean distance between the objects' current position and their closest medoid. Finding the optimal k medoids is NP hard, and existing algorithms aim at approximate answers, i.e., they compute medoids that achieve a small, yet not minimal, average distance. Similarly in this chapter, we also aim at approximate solutions. To the best of our knowledge, this work constitutes the first attempt on maintaining medoid synopses over moving objects streams. First, we consider *centralized* monitoring, where the objects issue location updates whenever they move. A server processes the stream of generated updates and constantly reports the current medoid synopsis. Next, we address *distributed* monitoring, where we assume that the objects have some computational capabilities, and they take over part of the monitoring task. In particular, the server installs adaptive filters (i.e., permissible spatial ranges, called *safe regions*) to the points, which report their location only when they move outside their filters. The distributed techniques reduce the frequency of location updates (and, thus, the network overhead and the server load), at the cost of a slightly higher average distance, compared to the centralized methods. Both our centralized and distributed methods do not make any assumption about the data moving patterns (e.g., velocity vectors, trajectories, etc) and can be applied to an arbitrary number of medoids k. We demonstrate the efficiency and efficacy of our techniques through extensive experiments.

For applications where the temporal dimension, e.g., recent history, of moving objects data streams is important we construct *spatiotemporal synopses*. We investigate the problem of maintaining hot motion paths synopses, i.e., routes frequently followed by multiple objects over the recent past. Motion paths approximate portions of objects' movement within a tolerance margin that depends on the uncertainty inherent in positional measurements. Discovery of hot motion paths is important to applications requiring classification/profiling based on monitored movement patterns, such as targeted advertising, resource allocation, etc. To achieve this goal,

we delegate part of the path extraction process to objects, by assigning to them adaptive lightweight filters that dynamically suppress unnecessary location updates and, thus, help reducing the communication overhead. We demonstrate the benefits of our methods and their efficiency through extensive experiments on synthetic datasets.

The remainder of this chapter is organized as follows. Section 5.1 motivates the need for moving object synopses and reviews related work. Section 5.2 describes two centralized methods for spatial synopses. Section 5.3 presents their adaption for a distributed setting. Section 5.4 adds the temporal dimension introducing the concept of trajectories and motion paths. Section 5.5 describes the RayTrace algorithm for filtering positional updates. Next, Section 5.6 presents the associated index structures for maintaining spatiotemporal synopses. Section 5.7.2 includes our experimental study. Finally, Section 5.8 summarizes this chapter.

5.1 Motivation and Related Work

To motivate the spatial k-medoid synopses consider a number of users accessing a location based service through their mobile devices, e.g., cellular phones or PDAs. To reduce the communication cost (and, thus, energy consumption), a number k of supernodes are selected among the mobile devices; the supernodes collect, aggregate and forward to the location server messages received from their vicinity. Due to signal attenuation for long distances, the devices should be close to some supernode. In other words, the supernode selection essentially reduces to a k-medoid computation over the set of devices. Additionally, the mobile nature of the system requires on-the-fly medoid maintenance. All the devices (supernodes or not) move frequently and arbitrarily, necessitating supernode re-assignment in order to retain the quality of service.

Regarding spatiotemporal synopses, consider the next motivating application. Assume a mobile phone carrier that wishes to serve targeted advertisements to subscribers. This service would be based on people's profiles and *continuous* market basket information about other clients that follow similar paths. For instance, in case of a major sporting event, many subscribers are expected to move towards the hosting venue. En route, a large number of them may stop by at certain facilities (e.g., rest areas, kiosks, malls) to purchase food, drinks, etc. This buying pattern (i.e., many people shopping for similar types of products at specific locations), can be utilized to promote a particular store that has an advertising deal with the mobile phone carrier. For example, customers passing by the advertised store during the event may be informed about its exact location and its current promotions or discounts.

Both examples discussed above motivate the on-line maintenance of spatial and spatiotemporal synopses summarizing the movement of multiple objects. We analyze their particularities in what follows.

Spatial synopses. Given a dataset P and a user-specified parameter k, the k-medoid spatial synopsis is a subset of P consisting of k points. These points are called the *medoids* and are selected so that the average distance between the points in P and their closest medoid is minimized. The k-medoid problem arises in many fields and application domains, including resource allocation, data mining, spatial

decision making, etc. Consider the example in Figure 5.1, where $P = \{p_1, ..., p_{24}\}$ is the set of residential blocks in a city, and fire stations are to be opened at three of them. To achieve the shortest average response time to emergency calls, we should minimize the average distance between residential blocks and their closest station. In this case, the best blocks to open fire stations at are the $k = 3$ medoids of P. In our example, the medoids are blocks p_6, p_{15} and p_{22}, shown in grey. The lines in the figure signify the assignment of the residential blocks to their responsible (i.e., closest) fire station. Due to this implicit assignment, k-medoids have also been used in different contexts for partitioning clustering.

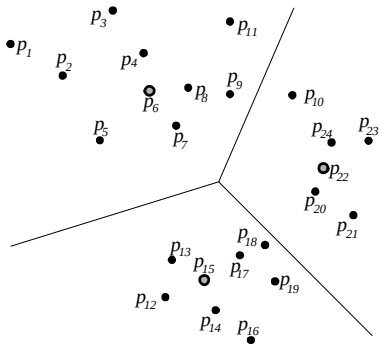


Figure 5.1: *A 3-medoid example*

Computing an optimal medoid synopsis is NP hard [31], and only approximate answers are possible even for relatively small input datasets. To this end, existing methods range from theoretical approximation schemes (e.g., [5]), to hill-climbing approaches for moderate size datasets (e.g., [55, 74]), to heuristic-based algorithms for disk-resident data (e.g., [28, 29, 71]). Focused also on disk-resident data, Mouratidis et al. [71] propose TPAQ, a method that solves k-medoid and related problems. All previous methods assume a static P, i.e., they compute the k medoids once and then terminate. In this chapter, we address a *dynamic* version of the problem, where the points in P send frequent location updates and the medoid set needs to be continuously maintained. In accordance with most real-world scenarios, the points in P move arbitrarily, with unknown motion patterns. We term the problem *continuous medoid monitoring*.

The existing k-medoid algorithms are unsuitable for our continuous monitoring setting. All aforementioned methods are designed for static datasets and snapshot queries (i.e., they compute the medoids once and then terminate); their extension to incremental medoid maintenance (in the presence of updates) is non-trivial, if possible at all. On the other hand, the naïve approach of re-computing from scratch the medoids (with some existing algorithm) in each update processing cycle is prohibitively expensive in a highly dynamic scenario, failing to reuse previous results. Additional problems of existing methods are: (i) the hill-climbing approaches are very slow for moderate or large input sizes, while (ii) TPAQ are designed for disk-resident data, with primary objective the minimization of the I/O cost; disk accesses are not an issue in our main memory setting, where CPU time (and communication cost, in the distributed case) is the only concern. On the other hand, an important finding of previous work to our problem is the efficiency and, more so, the efficacy of TPAQ, which motivates us to use a similar Hilbert-based (or, in general, space filling curve-based) approach for our purposes.

Regarding medoid-related problems in dynamic settings, Guha et al. [45] solve the k-medoid problem in a streaming environment. In the assumed model, the points of the input dataset P stream into the system. The main memory is not enough to store entire P, so the streamed data points are processed once and then discarded as new ones arrive. When the entire input set is seen, the system reports its k-medoids. [45] proposes an one-pass k-medoid algorithm that solves the above problem, using a small amount of space. Even though this is a dynamic method, it does not apply to our setting; in our case, (i) the memory does fit the *entire* dataset, but the points therein receive *location updates* in an on-line fashion, and (ii) the system needs to continuously report the k-medoid set *at any time*.

A problem related to k-medoids is *min-dist optimal-location* (MDOL) computation. The input consists of a set of data points P, a set of existing facilities (i.e., a set of existing medoids) and a user-specified spatial region R, wherein a new facility should open. The output of an MDOL query is the location in R where the new facility should be built in order to minimize the overall average distance between the data points and their closest facility. Zhang et al. [96] propose an exact method for this problem. The main differences from the k-medoid problem is that (i) MDOL assumes that a set of facilities already exists, (ii) it computes a single point (as opposed to k), and (iii) the returned point does not necessarily belong to P, but it can be anywhere inside region R.

The k-medoid problem is also related to clustering; essentially, given the medoids, the input dataset can be partitioned into k clusters by assigning each point to its closest medoid. The other direction, however, does not work; although there are numerous clustering methods for large input sets (e.g., DBSCAN [27], BIRCH [97], CURE [46] and OPTICS [4]), their objective is to create clusters such that the points in any cluster are more similar to each other than to points in other clusters. In addition to addressing a problem of different nature, most clustering algorithms are computationally intensive and unsuitable for the highly dynamic environments we tackle in this work.

In this chapter, we consider two system models corresponding to different mobile environments. First, we address *centralized* medoid monitoring. In this setting, the data objects[1] in P send updates to a central server whenever they move. The server processes the location updates and computes/reports the new medoid set. We propose two incremental monitoring algorithms that aim at minimizing the processing time for medoid maintenance. In the centralized model, the objects issue frequent location updates. This raises the additional concern about the communication cost. In particular, in many mobile computing applications, the objects have scarce power resources and we wish to preserve battery life by limiting the number of messages transmitted to the server. This motivates our second, *distributed* processing model. In this context, the server assigns *safe regions* to the data objects, which issue location updates only if they move outside their region. We design effective safe region computation strategies and incorporate them to our medoid monitoring framework. We demonstrate that the distributed methods drastically reduce the object communication overhead, while sacrificing minimal medoid quality (i.e., they result in marginally higher average distance compared to their centralized counterparts).

The first spatial monitoring techniques were targeted at range queries, where the data objects send location updates to a central server, and the latter continu-

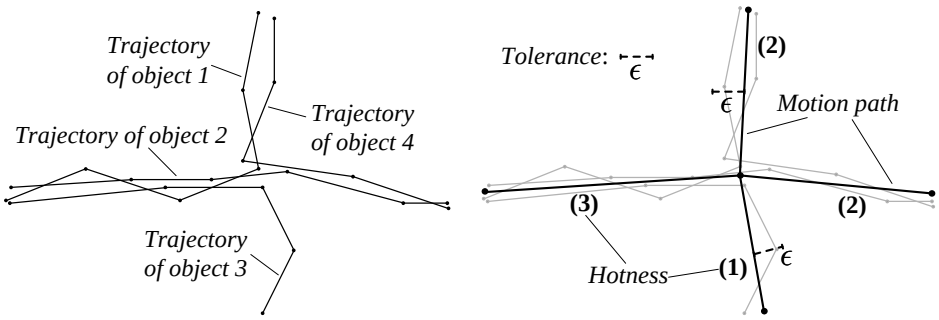
[1]Henceforth, the terms point and object are used interchangeably.

ously reports the objects that fall in each monitored range. *Q-index* [79] processes static range queries. It indexes the ranges using an R-tree and probes moving objects against the index in order to determine the affected queries and update their results. SINA [67] monitors (potentially moving) range queries using a three-step spatial join between moving objects and ranges. *Mobieyes* [35] and MQM [11] follow a distributed processing approach, where the objects utilize their computational capabilities and suppress some location updates. In particular, all of *Q-index*, *Mobieyes* and MQM utilize the concept of *safe regions*, according to which each object p is assigned a circular or rectangular region, such that p needs to issue an update only if it exits this area (because, otherwise, it does not influence the result of any query).

In addition to rage queries, several methods have been recently proposed for k Nearest Neighbor (k-NN) monitoring. Koudas et al. [56] present a system for approximate k-NN queries over streams of multidimensional points. Yu et al. [95], Xiong et al. [93] and Mouratidis et al. [69] describe algorithms for exact k-NN queries; all three methods index the data with a regular grid and maintain the k-NN results by considering only object movements that may influence some query. The aforementioned techniques aim at low processing time. There exist, however, methods designed for network cost minimization [70, 48] by exploitation of the objects' computational resources.

Spatiotemporal synopses. There are three challenging issues regarding the construction of spatiotemporal synopses representing the most important motion paths of multiple moving objects. First, numerous clients are expected to be on the move, so many object trajectories should be maintained. To enable effective decision making (e.g., advertising, alert scenarios), they need to be grouped and summarized, so that attention is drawn only to the most salient trails. We opt for a solution that consolidates multiple, neighboring trajectories into *motion paths* at the coordinator side. For each of them, we maintain its *hotness*, i.e., the number of objects that have recently traveled through it. Thus, end users are able to visualize/analyze only the hottest paths, and get a quick idea of the current situation. Figure 5.2 exemplifies this process, illustrating (a) the original object trajectories, and (b) the extracted motion paths and their hotness. As depicted, each motion path corresponds to a set of trajectory segments that evolve similarly, approximating them within a user-specified tolerance ϵ. In our framework, detected motion paths and their hotness are maintained in light-weight index structures enabling fast access. To restrict detection of salient paths to up-to-date readings, we impose a sliding time window of size W, which excludes from consideration any locations received more than W time units ago.

The second challenge in our design regards scalability in terms of communication overhead and computation cost at the coordinator. The naïve approach whereby all objects continuously relay their locations to the coordinator is practically infeasible because it incurs excessive bandwidth consumption, and may also lead to coordinator overloading due to the computational cost for motion path extraction. To alleviate these problems, we propose a distributed approach for processing and filtering location updates. Each object executes locally an algorithm (RayTrace) that compresses on-the-fly its trajectory abiding by a tolerance ϵ. Thus, the object itself reduces the number of locations that will be reported to the coordinator. Method RayTrace sets a permissible spatiotemporal extent, and transmits the recent trail only



(a) Input Trajectories (b) Motion Paths

Figure 5.2: *Motion path extraction*

when the current object location falls outside this filter. Aided by the coordinator, RayTrace then sets a new filter that reflects better its current motion pattern. This approach exploits the computational capabilities at the client side, while substantially reducing the communication overhead (due to fewer location updates) and the processing cost at the coordinator (for summarizing trajectories into motion paths). Such a setting is common in many sensor network applications and data streaming systems. For example, in [75, 18, 7] the coordinator aims to minimize communication cost by appropriately setting and updating filters on data sources that enable on-line calculation of counts, quantiles, and top-k entities, respectively. Filters of a spatial nature have been employed in the literature on continuous monitoring of range [36, 11, 80], nearest neighbor [48, 92] and medoid [77] queries.

Dealing with the inherent inaccuracy of location measurements is the third major consideration. Position readings are imprecise; moreover, they carry different degrees of uncertainty, depending on the handset capabilities and the network infrastructure. A GPS-enabled PDA provides more accurate location tracking than a cell phone, which relies solely on cellular triangulation for estimating its position. Furthermore, phones with just a few surrounding base stations offer less accurate measurements. Our proposed framework takes into account this uncertainty, as well as its varying degrees, and reports motion paths with discrepancy guarantees. Our handling of inaccuracy is aligned with the increasing interest in managing imprecise and uncertain data, such as in [9]. Closely related to our model of uncertainty, albeit not for spatio-temporal data, is the work in [86] that proposes an index for storing and querying imprecise spatial locations modeled by some probability density function.

The problem of discovering frequently followed, i.e., hot, routes has also been examined in a recent work [58], but only for the case that objects are confined to move in a known network. A hot route in this context is a sequence of edges, not necessarily adjacent, that share a high amount of traffic. The approach presented in this chapter differs in several aspects. First, it assumes unrestricted movement on the xy plane; second, in measuring hotness it considers the time interval that objects crossed each designated path; and, third, it accounts for imprecision in positional measurements.

Our work is relevant to the domain of spatiotemporal data reduction, particularly to the topic of trajectory compression. Most existing algorithms [12, 66] attempt to compress singleton trajectories in isolation, by adapting the off-line Douglas-Peucker algorithm [26]. This line simplification technique drops the least important vertices

to achieve a reduced representation. It is widely used in spatial databases, but it requires multiple passes over the data, which yields it inapplicable to on-line streaming applications. Sampling techniques have been proposed in [78] for compressing isolated trajectories. Furthermore, segmentation of multiple trajectories by fitting into axis-parallel rectangles is considered in [3, 47], where dynamic programming, greedy and heuristic techniques are employed so as to minimize the empty space in rectangles or to preserve pair-wise distances among trajectories.

An adaptation of Douglas-Peucker algorithm more suitable to dynamic environments was presented in [66]. Instead of considering the entire trace of an object for applying line generalization, an opening window principle is employed to reduce the amount of timestamped locations considered at each step. The technique starts processing positions in temporal order and progressively produces successive line segments. More specifically, after fixing a starting point, the algorithm examines candidate line segments by setting their floating endpoint as much farther as possible, provided that all intermediate locations are within a given tolerance from the constructed segment. In case this rule is violated, two alternative policies were proposed for fixing the endpoint of the new segment. The conservative approach (DP-nopw) chooses the location that caused the violation, i.e., the one with the greatest distance from the examined segment. The eager approach (DP-bopw) takes the location with the greatest possible timestamp, which is the one just before the floating endpoint. Checking violations is very costly, since all locations between the starting point and the current floating endpoint must be examined each time. Overall, this method is constrained to choose a subset of the reported locations as endpoints and thus, it offers a rather strict trajectory synopsis. In Section 5.7.2 we describe an adaptation of this technique to hot motion path computation and use it as a competitor.

Detecting clusters of moving objects, moving clusters and frequent motion patterns has also attracted research interest. Clustering similar objects based on their movement characteristics, e.g., current position and velocity, is discussed in [59, 52]. More related to our problem is the work in [53] for identifying dense clusters of objects which move similarly over a long period of time. According to their definition clusters need not contain the same set of objects all along their lifetime. The difference from the problem we tackle here is twofold. First, although moving clusters evolve across a path that is interesting (i.e., hot), we only need to identify and maintain the motion paths per se, and not the actual clusters or their constituent objects. The second, and most important, reason is that a motion path may be hot even when no moving cluster crosses it. To justify this, note that a moving cluster requires objects to be close enough to each other at *any time instant* during a sliding window of W time units. In contrast, a motion path can become important as long as a sufficient number of objects have crossed it in the last W time units, no matter if they travel synchronously or not across that path. The work in [60] computes spatial regions containing frequent periodic (e.g., daily, weekly, etc) motion patterns. Besides limitations including a priori knowledge of periodicity, this method treats trajectories merely as sequences of locations (i.e., it eliminates timestamps), hence, being inapplicable to our timestamp-sensitive problem.

Another topic related to our work is trajectory clustering. From a data mining perspective, a methodology was introduced in [30] and it was based on a probabilistic mixture of regression models, which the moving objects are assumed to follow.

Also, detecting similarity among trajectories or timeseries has also attracted considerable research interest. Common problems have to do with outliers, local shifts in time, as well as movements of varying total length. Several distance measures have been proposed in order to identify similar trajectories or subsequences, e.g., Time Warping Distance [94], Longest Common Subsequence [90], and Edit Distance on Real Sequences [15]. However, all these techniques are not particularly tailored to handle on-line trajectories, since they require comparisons over large portions of objects' movement, while their objective is to group together trajectories in their entirety. Hence, they fail to identify trajectories that locally follow common routes, because the overall computed distance is greatly affected by distant segments. The most recent approach was presented in [57] and proposes a technique for clustering smaller linear partitions instead of entire trajectories stored in a database. The main idea of the algorithm is that trajectories are first split into several parts at characteristic points and then similar line segments are grouped together into a cluster. For identifying common motion patterns, the minimum description length (MDL) measure is adapted from the domain of pattern recognition, but it seems quite sensitive to appropriate selection of parameters. Moreover, time is ignored and trajectory segments are considered to be spatial polylines.

In conclusion, the contributions of our work can be summarized as follows.

1. We introduce two algorithms, HBM and GBM, for maintaining spatial k medoid synopses over moving object streams.

2. We extend the aforementioned algorithms to a distributed environment applying the concept of safe areas.

3. We propose a spatiotemporal reduction algorithm, RayTrace, for compressing the uncertain trajectory of a moving object.

4. We propose the SinglePath algorithm for extracting spatiotemporal motion path synopses for a large set of moving objects.

5. We present extensive experimental results of our algorithms on synthetic datasets. Our experimental study demonstrates: (i) the applicability of HBM and GBM, and their distributed counterparts, for high rate streams; (ii) the importance of motion paths as a spatiotemporal synopsis method; and (iii) the efficiency of SinglePath in discovering important motion paths.

5.2 Centralized Spatial Synopses

In this section we present our centralized methods. We assume that dataset P consists of $|P|$ two-dimensional points. Although our methods are applicable to higher dimensions, in accordance with most real-world mobile environments, we focus on two dimensions. Furthermore, for ease of presentation, we consider a unit data space, i.e., all data fall in $[0,1]^2$. Every point p in P is a tuple of the form $< p.id, p.x, p.y >$, where $p.id$ is a unique identifier and $(p.x, p.y)$ are p's coordinates. Whenever p moves, it issues an update to the monitoring server; the update has the

form $< p.id,\ p.x_{old},\ p.y_{old},\ p.x_{new},\ p.y_{new} >$[2], implying that p moves from ($p.x_{old}$, $p.y_{old}$) to ($p.x_{new}$, $p.y_{new}$). The objects move frequently and arbitrarily.

We present two centralized medoid monitoring algorithms, based on a common intuition exemplified in Figure 5.3. Dataset P contains two clusters C_1 and C_2. Suppose that a 2-medoid query returns one medoid in C_1 and another in C_2. Now consider that we wish to compute three medoids. Observe that, although C_1 has a smaller *diameter* than C_2, it contains more points. Due to the larger cardinality of C_1, the distances of its points from its medoid affect the global average distance to a greater extent than that of the points in C_2. Therefore, placing the third medoid in C_1 leads to a larger distance reduction than placing it in C_2. Intuitively, more medoids must be assigned to denser areas of the data space.

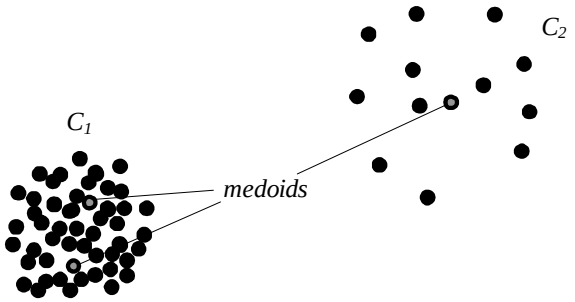


Figure 5.3: *The three medoids of a dataset consisting of two clusters*

Motivated by this observation, our algorithms (i) partition the points in P into k groups of (roughly) equal cardinality and, then, (ii) select the most centrally located object from each group as the corresponding medoid. To quickly perform step (i) we project the points on a one-dimensional space using a space filling curve. We employ the Hilbert curve since it is shown to best preserve locality compared to alternatives [68]. Next, we partition the Hilbert-sorted list of points into k groups of equal cardinality (i.e., $|P|/k$). Due to the locality preservation of the Hilbert curve, the resulting groups can be regarded as well-defined partitions of P in the two-dimensional space. Finally, we extract a medoid from each group; the medoid is the point in the group with the median Hilbert value, as it is expected to be the most centrally located. The above rationale underlies both modules of our algorithms, namely, the initial medoid computation and their maintenance.

5.2.1 The HBM Algorithm

Our first method is *Hilbert-based Monitoring* (*HBM*). It indexes the data objects with an in-memory 2-3 B^+-Tree [16] (i.e., a B^+-Tree where each internal node has two or three children), using their Hilbert values as search keys. We denote this tree by BT. At the leaf level, except for the standard *right sibling* pointers, BT is modified to also accommodate *left sibling* pointers. In other words, the leaves are organized as a doubly connected linked list. When the continuous medoid query is installed at the server for the first time and BT is built, every entry E in an internal node N *temporarily* stores aggregate information about the number of points

[2]If the update is an insertion (deletion), $p.x_{old}$, $p.y_{old}$ ($p.x_{new}$, $p.y_{new}$) are set to a negative value.

$E.a$ contained in its subtree. $E.a$ facilitates the initial medoid computation and is discarded afterwards.

In particular, according to our general approach, the i-th medoid of P is the $[(i\text{-}0.5)\cdot |P|/k]$-th object in the linear order imposed by the Hilbert values. HBM locates the k medoids by performing k traversals in BT, at a total cost of $O(k\cdot \log|P|)$. Before each traversal i, an auxiliary variable V is initialized to zero. The traversal starts from root N_R and it checks whether $V + E_1.a$ is larger than or equal to $(i\text{-}0.5)\cdot |P|/k$, where E_1 is N_R's first entry. If that is the case, the medoid is located in E_1's subtree and, therefore, the traversal continues by visiting E_1's child. Otherwise, $E_1.a$ is added to V and the algorithm continues similarly by checking $V+E_2.a$ against $(i\text{-}0.5)\cdot |P|/k$ (E_2 is N_R's second entry). Valways keeps the number of points preceding (in the Hilbert order) the point with the smallest search key that is reachable by the traversal. Finally, the algorithm reaches the leaf node containing the i-th medoid. For every computed medoid m, an array M of size k stores a tuple of the form $< m.id, m.hv, m.ptr, m.off >$, where $m.id$ is the identifier of the point selected as m, $m.hv$ is m's Hilbert value, $m.ptr$ points to the leaf node of BT that accommodates m, and $m.off$ is an integer (initialized to zero) used by the maintenance module and whose functionality is explained later. The temporary $E.a$ values are discarded after the end of the initial computation step. Figure 5.4 summarizes the data structures in HBM.

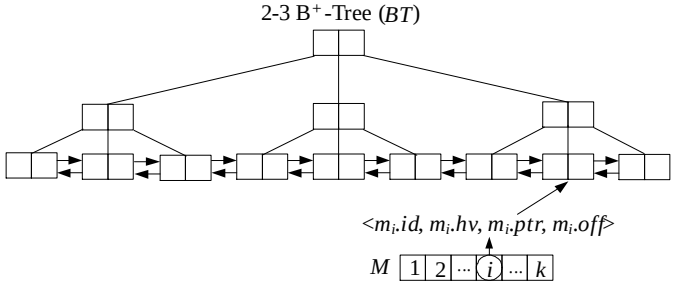


Figure 5.4: *The data structures of the HBM method*

The server periodically receives updates from the objects in batches. HBM accordingly updates BT, after computing the necessary Hilbert values of the inserted, deleted or moving points. Note that the movement of an object involves its deletion from the index followed by its subsequent re-insertion with the new Hilbert value. Whenever a split or merge operation moves a medoid m to a different leaf node, the corresponding $m.ptr$ must also be altered in M. While updates are reflected in BT, HBM stores some book-keeping information, to be used for result maintenance according to its medoid selection strategy. In particular, after processing the insertion/deletion of a point p, HBM performs a binary search in array M to locate the leftmost medoid m_u with Hilbert value greater than (or equal to) $p.hv$. In case p initiated an insertion (deletion), the algorithm increases (decreases) $m_u.off$ by one. Particular care must be taken when a medoid m is deleted. In this case, HBM substitutes it with its predecessor in the Hilbert order and decreases $m.off$ by one.

After processing all updates, HBM computes the new medoids as follows. The i-th medoid m_i was formerly data point p_{old} at position $(i\text{-}0.5)\cdot |P|/k$. After the updates, p_{old} moves to position $(i\text{-}0.5)\cdot |P|/k + \sum_{j=1}^{i-1} m_j.off$. The actual medoid must be located at position $(i\text{-}0.5)\cdot |P|/k$, where P is the updated version of dataset P (which may have different cardinality if new objects were inserted or existing ones

Function **updateMedoids** (*array M, Tree T*)

1. Initialize V to 0
2. For i=1 to k
3. Locate medoid m_i in leaf $M[i].ptr$ of T
4. OFF_i = (i-0.5)· $|P|/k$- (i-0.5)· $|P|/k - V$
5. If $OFF_i == 0$, continue
6. Else if $OFF_i > 0$, find point p located $|OFF_i|$ positions to the right of m_i
7. Else if $OFF_i < 0$, find point p located $|OFF_i|$ positions to the left of m_i
8. V += $M[i].off$;
   (a) Assign $p.id$, $p.hv$, the pointer of the leaf of T that accommodates p and 0 to $M[i].id$, $M[i].hv$, $M[i].ptr$ and $M[i].off$, respectively

Figure 5.5: *The maintenance module of HBM*

deleted). Therefore, the new medoid m_i can be found OFF_i = (i-0.5)· $|P|/k$- (i-0.5)· $|P|/k$ - $\sum_{j=1}^{i-1} m_j.off$ positions to the right or left of p_{old} in the linear order, depending on whether OFF_i is positive or negative, respectively. For every medoid m_i in M, HBM first visits the leaf node pointed by $m_i.ptr$ to find its old corresponding point p_{old}. Then, using the left/right sibling pointers of BT, it locates the new medoid and properly updates m_i's entry in M. The pseudocode of the maintenance procedure is given in Figure 5.5.

Figure 5.6 illustrates the initial computation and maintenance of $k = 2$ medoids in a set of points, which at timestamp T_1 has cardinality 14. For ease of demonstration, we omit the BT operations and focus on the leaf level of the tree, which constitutes a doubly connected linked list of points sorted on their Hilbert values. At timestamp T_1, the set is subdivided into two subsets of seven points each. The medians of the subsets (p_4 and p_{12}) are selected as the medoids (m_1 and m_2, respectively). At timestamp T_2, four updates occur; p_1 and p_{13} are deleted, and p_3 and p_5 move to new positions. Due to p_1's deletion, $m_1.off$ is decreased by one. On the contrary, the deletion of p_{13} does not affect any *off* value because there is no medoid with higher (or equal) Hilbert value. Regarding p_3 and p_5, recall that a point movement is handled as a deletion followed by an insertion. Upon p_3's deletion, the algorithm decreases $m_1.off$. Subsequently, the point is re-inserted in a position between m_1 and m_2 and, therefore, $m_2.off$ is increased by one. Finally, p_5's movement causes $m_2.off$ to decrease (due to its deletion) and immediately increase (due to its re-insertion) by one, because both its old and new Hilbert values are between $m_1.hv$ and $m_2.hv$. Let old_pos_i be the position (in the Hilbert order) of the point that was selected as medoid m_i at timestamp T_1. Also let $curr\_pos_i$ be the position of the new point to become m_i at timestamp T_2. For m_1, $old\_pos_1 = 4$, $curr\_pos_1 = 3$, and $OFF_1 = 1$. Similarly for m_2, $old\_pos_2 = 11$, $curr\_pos_2 = 9$, and OFF_2 = -1. The algorithm locates the new medoids p_3 and p_{11}, by moving one position to the right

and one to the left from old medoids p_4 and p_{12}, respectively.

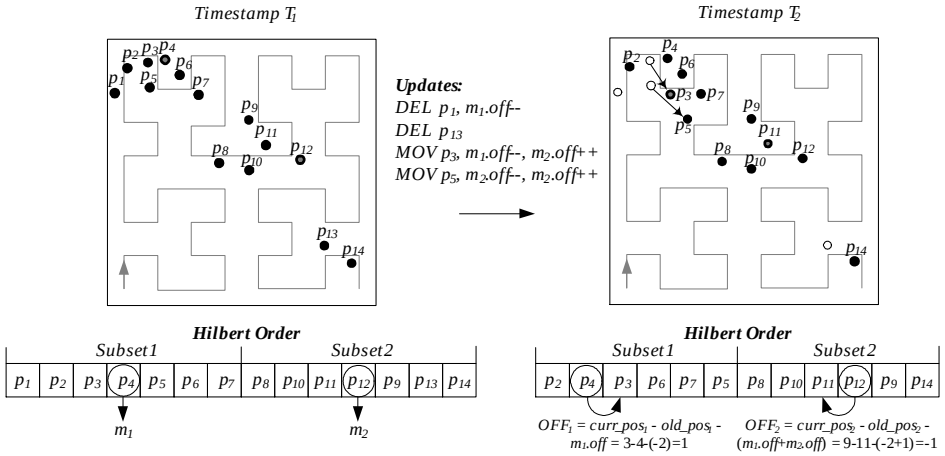


Figure 5.6: *A medoid monitoring example in HBM*

5.2.2 The GBM Algorithm

The *Grid-based Monitoring* (*GBM*) algorithm utilizes a $C \times C$ regular grid for indexing P. Let δ be the side-length of each cell. A point p in P with coordinates $(p.x, p.y)$ can be located in constant time in cell $c_{i,j}$ (i.e., the cell in column i and row j, starting from the low-left corner of the grid), where $i = \lfloor p.x/\delta \rfloor$ and $j = \lfloor p.y/\delta \rfloor$. GBM imposes a linear order on the cells by sorting them according to the Hilbert values of their centers. Every cell c is associated with a tuple $< c.n$, $c.prev$, $c.next$, $c.BT >$, where $c.n$ is the cardinality of the set of points contained in c, $c.prev$ and $c.next$ are the cells preceding and succeeding c in the Hilbert order respectively, and $c.BT$ is a BT that indexes the points in c (using their Hilbert values as search keys). Similarly to HBM, the internal nodes in the BTs temporarily incorporate aggregate information, which is discarded after the initial computation of the medoids.

The grouping strategy of GBM is similar to HBM, the difference being in the linear order of the points, which now takes into account firstly the order of the cells. Specifically, the points are considered sorted according to the following rules; (i) a point p_1 in cell c_1 precedes point p_2 in cell c_2, if c_1 precedes c_2 in their Hilbert order, and (ii) the order of the points in the same cell is determined by their Hilbert values. Following similar reasoning as in HBM, the i-th medoid m_i is the $[(i\text{-}0.5)\cdot |P|/k]$-th object in the above order. GBM starts by initializing an auxiliary variable V to zero and scans the linked list of the (sorted) cells. To locate medoid m_i, in every visited cell c_i, it checks whether $V + c_i.n$ is larger than or equal to $(i\text{-}0.5)\cdot |P|/k$. If that is the case, it traverses $c_i.BT$ in order to find the $[V + c_i.n\text{-}(i\text{-}0.5)\cdot |P|/k]$-th object in the cell, which is then selected as medoid m_i. Otherwise, it adds $c_i.n$ to V and continues to the next cell. V keeps the number of points encountered by the scan so far. Note that GBM locates all medoids in a single linear scan of the cells, i.e., after finding medoid m_i, it does not restart the scan for finding m_{i+1}; instead, it continues from the cell that contains m_i. Finally, it maintains an array M with functionality identical to that used by HBM. Figure 5.7 depicts the data structures of GBM.

For every received update, GBM first determines in constant time the cell c where the insertion/deletion takes place, and properly updates $c.BT$. Subsequently,

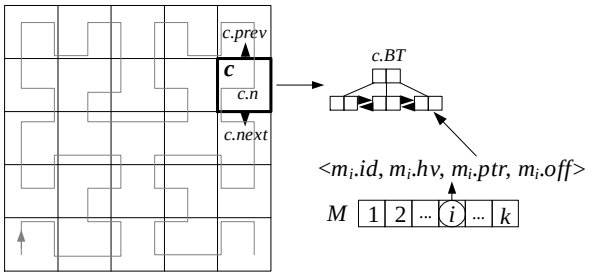


Figure 5.7: *The data structures of the GBM method*

it scans M and updates the *off* value of the leftmost medoid with Hilbert value larger than or equal to that of the object that initiated the update, in a similar fashion to HBM. After processing all the updates, the maintenance module of GBM identifies the points to be selected as the new medoids as follows. It scans M and for every m_i, it computes OFF_i in a fashion similar to Section 5.2.1. Suppose that m_i lies in cell c. Then, starting from the leaf of $c.BT$ that accommodates m_i and is pointed by $m_i.ptr$, it searches for the point that will be selected as the new m_i. This point lies OFF_i positions to the left or right of old m_i, depending on whether OFF_i is negative or positive, respectively. If the search reaches the leftmost or rightmost (in the Hilbert order) point of cell c, it continues to the cell pointed by *c.prev* or *c.next*, respectively. Note that the algorithm may skip entire cells (i.e., it may not traverse their BTs at all), since it can always determine whether m_i is located in a visited cell by comparing the cell's cardinality against OFF_i. After finding a new medoid, GBM updates the respective entry in M accordingly.

In Figure 5.8 we exemplify the initial medoid computation and monitoring in a scenario where $k = 2$ and P contains points p_1 to p_{14}. Consider cells $c_{2,2}$ and $c_{1,2}$ (a)t timestamp T_1. The Hilbert curve first passes through $c_{2,2}$ and, thus, p_{11} precedes p_1 in the GBM order, although it succeeds it in the global Hilbert order (i.e., $p_{11}.hv > p_1.hv$, where $p_{11}.hv$ and $p_1.hv$ are the Hilbert values of p_{11} and p_1, respectively). At timestamp T_1, the medoids are $m_1 = p_4$ and $m_2 = p_{14}$, since they are at positions $0.5 \cdot |P|/k = 4$ and $1.5 \cdot |P|/k = 11$, respectively, in the linear order. At timestamp T_2, objects p_7, p_6 and p_{11} issue updates, as shown in the figure. Their movement leads to $OFF_1 = 1$ and $OFF_2 = 1$, and updates the medoids to $m_1 = p_7$ and $m_2 = p_{11}$.

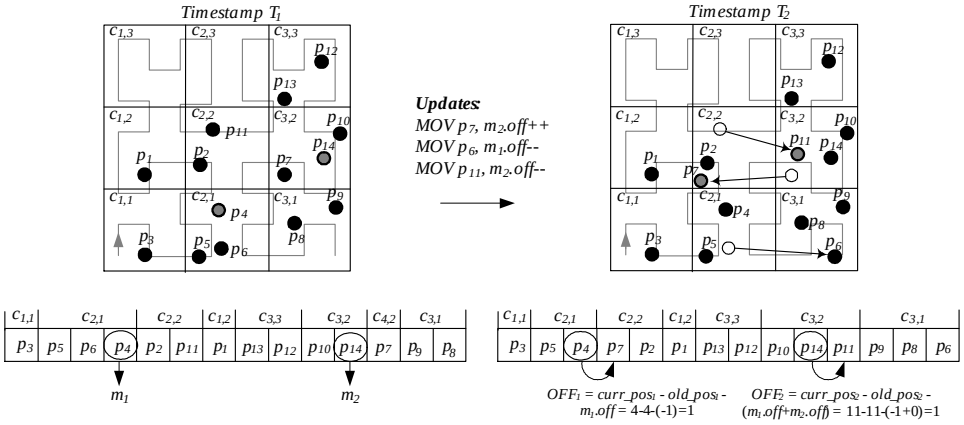


Figure 5.8: *A medoid monitoring example in GBM*

Compared to HBM, index update and medoid maintenance in GBM are expected to be faster. HBM keeps a common BT over all $|P|$ points, which leads to an $O(\log|P|)$ cost for every point insertion or deletion. On the other hand, letting c be the cell of the inserted/deleted point, $c.BT$ contains $c.n$ objects (where $c.n \ll |P|$), requiring $O(\log|c.n|)$ time per update. Furthermore, maintaining the medoids is also more efficient in GBM, because for large OFF_i values, entire cell contents may be skipped when sliding in the linear point order towards the new medoid position. Another major advantage of GBM over HBM, is the fact that its data index is compatible with existing methods for other spatial query types; most range and nearest neighbor monitoring algorithms use a regular grid index. This allows GBM to be used in conjunction with other methods, in a system that answers general spatial queries over moving objects, utilizing a single data index.

A final remark concerns the average distance, which is in general different but similar for GBM and HBM, since their medoid selection rationale is alike. In particular, if the grid granularity in HBM is selected so that C is a power of two (recall that the grid has $C \times C$ cells), their medoids are identical. The reason is that the Hilbert values themselves are computed by definition based on a transparent space partitioning with a grid, whose granularity on each axis is always a power of two (this power is called the *order* of the Hilbert curve). If C is also a power of two, the cells of the object grid contain continuous, non-overlapping intervals of the curve. In other words, if cell c_1 precedes c_2 on the curve, then *any* point p_1 in c_1 precedes *every* p_2 in c_2. In turn, this fact implies that the linear point orders of GBM and HBM are identical and, thus, the medoids are the same.

5.3 Distributed Spatial Synopses

The main idea in the distributed version of our methods is to allow objects to move within assigned *safe regions*, without having to transmit updates to the server. Since our general medoid selection strategy relies on a linear point order, the safe regions are defined with respect to the neighboring objects (in the order). Particularly, let *leeway* λ be an integer system parameter. The safe region of the i-th object in the order p_i is a Hilbert interval $SR_i^\lambda = [p_i.sr_L,\ p_i.sr_R]$. The left boundary $p_i.sr_L$ is the mean of the Hilbert values of p_i and its λ-th left neighbor $p_{i-\lambda}$ (i.e., $p_i.sr_L = \lfloor (p_i.hv + p_{i-\lambda}.hv)/2 \rfloor$). The right boundary $p_i.sr_R$ is set similarly with respect to the λ-th right neighbor (i.e., $p_i.sr_R = \lceil (p_i.hv + p_{i+\lambda}.hv)/2 \rceil$). Object p_i may change location without issuing an update, as long as $p_i.hv \in$ $((\lambda)())$. When p_i does move outside SR_i^λ, it sends its new location to the server. The latter updates its index and the medoid set accordingly[3], and assigns a new safe region to p_i. Note that the new SR_i^λ is defined based on the latest point positions reported. Particularly for GBM, the linear point order takes into account the grid cells ordering. Thus, the safe regions are defined within each cell individually (i.e., in the Hilbert order of the objects therein). Whenever an object exits its cell, it sends an update regardless of whether it violates its safe region.

Figure 5.9 demonstrates the safe region function in the case of HBM (the case of GBM is similar, subject to the aforementioned modifications), showing the position of the points on the Hilbert curve. At timestamp T_1, the safe region $SR_3^{\lambda=2}$ ($SR_3^{\lambda=2}$)

[3]Medoid maintenance at the server side is identical to the centralized case.

of p_3 is defined according to p_2 and p_4 (p_1 and p_5) for $\lambda = 1$ ($\lambda = 2$). Similarly, $SR_4^{\lambda=1}$ is determined by p_3 and p_5. Assuming that $\lambda = 1$, at timestamp T_2, points p_3 and p_4 move. However, only p_3 issues an update, because p_4 remains within its safe region. The solid points in the figure correspond to the positions known by the server, the hollow point is p_3's old Hilbert value, while the grey is p_4's actual one. Object p_3 is assigned a new region, based on the Hilbert values of p_2 and p_4. Note that the server is not aware of the new location of p_4 and, thus, uses the last reported one (as of T_1).

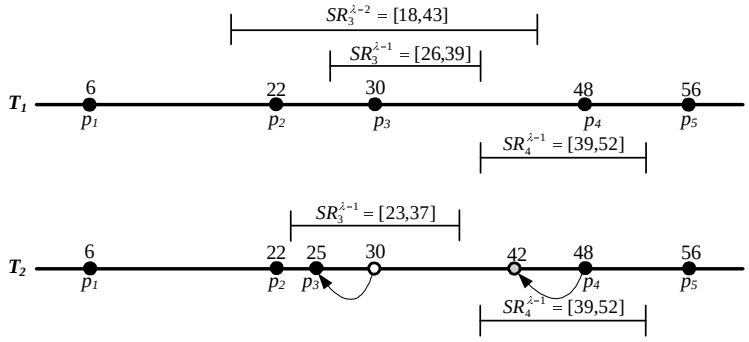


Figure 5.9: *Safe regions and update handling*

5.4 Trajectories and Motion Paths

We consider objects moving in the xy plane and hence all spatial locations are points $\mathbf{p}_i = (x_i, y_i)$. A point $\mathbf{p}_i$ accompanied with a timestamp t_i is called a *timepoint* and denoted as $\langle \mathbf{p}_i, t_i \rangle$. The *trajectory* of an object consists of a set of timepoints $\mathbf{T} = \{\langle \mathbf{p}_i, t_i \rangle\}$. The location of an object at time t_i is denoted as $\mathbf{T}(t_i) = \mathbf{p}_i$. Following common practice, between any two consecutive timestamps the object is assumed to move with constant velocity. As a result, the object's location at time t_k, where t_i, t_{i+1} are consecutive timestamps and $t_i < t_k < t_{i+1}$, is considered to lie in the (directed) segment $\mathbf{p}_i\mathbf{p}_{i+1}$ and can be calculated using linear interpolation. In the following, we assume that time is discrete and that all timestamps are multiples of some time granule.

We say that a point $\mathbf{p}_a$ is *close* to an object with trajectory $\mathbf{T}$ if there exists a time t_k such that $\mathbf{p}_k = \mathbf{T}(t_k)$ is within distance ϵ to $\mathbf{p}_a$, where ϵ is a user-specified tolerance parameter. In other words, the object has passed near $\mathbf{p}_a$ at some time t_k. Even though our methods apply to any L_p metric (including the Euclidean), for ease of illustration in the following we assume the max-distance, i.e., the distance between $\mathbf{p}_a$ and $\mathbf{p}_k$ is defined as $\max\{|x_a - x_k|, |y_a - y_k|\}$. Given tolerance ϵ, a directed line segment $\mathbf{p}_a\mathbf{p}_b$ is called a *motion path* if there exists a time interval $[t_a, t_b]$ such that point $\mathbf{p}(\lambda) = \mathbf{p}_a + \lambda(\mathbf{p}_b - \mathbf{p}_a)$ is close (within distance ϵ) to some object's location $\mathbf{T}$ at time $t(\lambda) = t_a + \lambda(t_b - t_a)$ for all $\lambda \in [0, 1]$[4]. We say that the object *crosses* the motion path and, inversely, that the motion path *fits* the object's movement. Intuitively, an object traveling during time interval $[t_a, t_b]$ along motion path $\mathbf{p}_a\mathbf{p}_b$ would always be within distance ϵ to another object.

Figure 5.10 draws with bold line the trajectory of an object moving along the x axis versus time t. The shaded envelope represents all points that are within

[4]Since time is discrete, the λ values are selected so that $t(\lambda)$ is a valid timestamp.

distance ϵ to the trajectory (at some timestamp). Figure 5.10 also shows 4 motion paths, $\mathbf{p}_a\mathbf{p}_b$, $\mathbf{p}_c\mathbf{p}_d$, $\mathbf{p}_e\mathbf{p}_f$, $\mathbf{p}_g\mathbf{p}_h$. The object crosses these motion paths during the time intervals, $[t_a, t_b]$, $[t_c, t_d]$, $[t_e, t_f]$, $[t_g, t_h]$, respectively. A motion path paired with its associated time interval draws a line segment on the xt plane that is completely inside the shaded envelope.

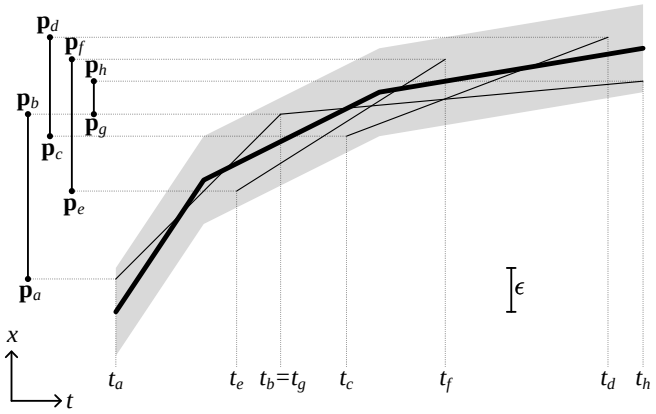


Figure 5.10: *Motion paths example*

Note that for a single object and for any time interval one could find an infinite number of motion paths. Fix some object i, and let $S_i = \{\langle \mathbf{p}_a\mathbf{p}_b, t_at_b\rangle\}$ denote a set of pairs consisting of a motion path $\mathbf{p}_a\mathbf{p}_b$ that fits the object's movement together with the time interval $[t_a, t_b]$ during which the object crosses it. We say that S_i is a *covering* motion path set for object i if at any time t_k the object either crosses a single motion path, or crosses two motion paths $\mathbf{p}_a\mathbf{p}_b$, $\mathbf{p}_c\mathbf{p}_d$, but $t_k = t_b = t_c$ and $\mathbf{p}_b \equiv \mathbf{p}_c$, i.e., one's start point is the other's end point. A covering motion path set implies that one could construct a hypothetical object whose trajectory is always close to object i's trajectory. For this reason a covering set can be considered as a simplification of the object's movement. A motion path is considered *valid* if it belongs to a covering motion path set for some object. In the remainder of this chapter, we only deal with valid motion paths and, thus, omit the valid denotation. Returning to the example in Figure 5.10, $S = \{\langle \mathbf{p}_a\mathbf{p}_b, t_at_b\rangle, \langle \mathbf{p}_g\mathbf{p}_h, t_gt_h\rangle\}$ is a covering motion path set for the object considered. Indeed, $\mathbf{p}_b \equiv \mathbf{p}_g$ and $t_b = t_g$.

In the previous, we have assumed that the object's location is accurately known. In a more realistic setting, though, the location sensing device reports coordinates with a degree of spatial uncertainty. The position of an object constitutes a random vector $\mathbf{P}_i = (X_i, Y_i)$, where X_i, Y_i are independent random variables. Let us note that there is no uncertainty regarding the timestamp. We repeat the previous definitions considering spatial uncertainty. Given tolerance ϵ and δ, we say that a point $\mathbf{p}_a$ is *close* to an object with trajectory $\mathbf{T}$ if there exists a time t_k such that $\mathbf{P}_k = \mathbf{T}(t_k)$ is within distance ϵ to $\mathbf{p}_a$ with probability greater than $1-\delta$. Assuming the max-distance metric, we require:

$$Pr\left(\max\{|X_k - x_a|, |Y_k - y_a|\} \leq \epsilon\right) \geq 1 - \delta.$$

The motion path in the presence of spatial uncertainty is defined accordingly, considering the aforementioned definition of proximity.

Recall that a motion path could fit multiple objects (or even the same object) during different time intervals. We define the *hotness* of a motion path to be the

number of times objects have crossed it during the past W time units. The problem of hot motion path discovery can be formulated as follows:

Problem 5.1 [Hot Motion Paths] For a set of moving objects, given tolerance ϵ (or ϵ, δ) and a time window of length W, find covering motion path sets and report the top-k hottest motion paths. □

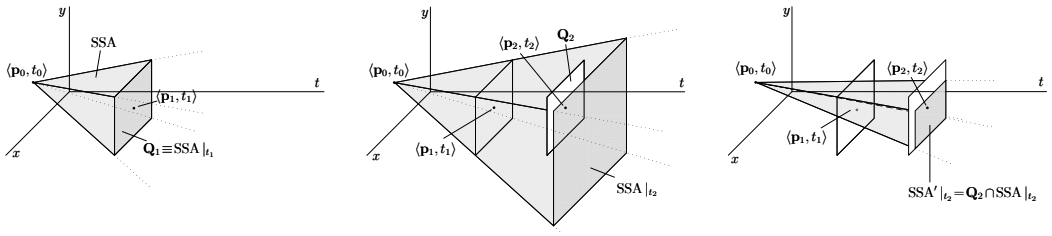
Intuitively, Problem 5.4 states that we wish to discover motion paths that are crossed frequently by many objects. Depending on the chosen covering motion path sets, the characteristics of the top-k hottest motion path can vary greatly. Since this problem is motivated by the need to identify generalized frequent flows of movement, the best top-k result should ideally contain motion paths that are as large as possible (abiding by the tolerance parameters) and as hot as possible. Hot and large motion paths clearly convey more information (e.g., objects have crossed them and stayed close to each other for a long time), compared to short, but equally hot paths. To assess the quality of the top-k hottest motion paths, we devise a simple metric, termed *score*, that promotes longer paths. The score of a motion path is defined as its hotness multiplied by its length, and the score of the top-k set is the average score of its motion paths.

Given this notion of quality, the discovery process set forth in Problem 5.4 requires us to carefully construct long motion paths so that they fit as many objects as possible. Considering the freedom in choosing covering motion path sets for each object, this clearly becomes a daunting task. To emphasize on the latter, consider the case of a single moving object. Problem 5.4 degenerates to summarizing the object's trajectory with the fewest, and hence longest, segments. The solution [49] to this degenerate case requires two passes over the timepoints and requires linear space and time. As we discuss in the next section, such algorithms are prohibitive in our setting since they require storing all timepoints seen so far. Before proceeding to this section and the system model description, we summarize in Table 5.1 definitions and notation used throughout this chapter.

Symbol	Description
$\mathbf{p}_i = (x_i, y_i)$	point in xy space
$\langle \mathbf{p}_i, t_i \rangle$	timepoint in xyt space
$\mathbf{T}_i = \{\langle \mathbf{p}_j, t_j \rangle\}$	trajectory of object i
ϵ, δ	tolerance parameters
$\langle \mathbf{s}^i, t_s^i \rangle$	start of a motion path for object i
$\langle \mathbf{e}^i, t_e^i \rangle$	end of a motion path for object i
Tolerance Square	square of side 2ϵ around point $\mathbf{p}_j$
Spatial Safe Area (SSA)	pyramid $(\mathbf{l}^i(t), \mathbf{u}^i(t))$ in xyt space
Final Safe Area (FSA)	rectangle $(\mathbf{l}^i, \mathbf{u}^i)$ at time t_e
$State_i$	the state transmitted to coordinator
W	time window
Λ	processing epoch
h_j	hotness of motion path $\mathbf{p}_j\mathbf{p}_{j+1}$
$\mathcal{AP}_i$	available motion paths for object i
$\mathcal{CP}_i$	candidate motion paths for object i
$\mathcal{AV}_i$	available vertices for object i
$\mathcal{CV}_i$	candidate vertices for object i

Table 5.1: *Primary symbols and functions*

We consider an environment where the moving objects are geographically distributed and can communicate with a central coordinator. Each object is capable of sensing its own location with some uncertainty (modeled by tolerance ϵ, δ) and is capable of performing simple processing tasks requiring little memory. In this



(a) $t = t_1$ (b) $t = t_2$, before (c) $t = t_2$, after

Figure 5.11: *Updating the SSA*

setting, the coordinator must maintain hot motion paths by collecting information from the objects.

There are two main issues we must take into account in this setting. First, objects have scarce battery life. Sending messages over the communication channel is typically orders of magnitude more power consuming compared to CPU processing. Following common practice, we must strive to minimize communication to and from the coordinator. Furthermore, objects listen for incoming messages only at predefined time instances termed *epochs*, i.e., every Λ time units. The second issue is the streaming nature of location measurements. An object should not store the unbounded stream of measurements, let alone transmit it to the coordinator; rather, it should only store information necessary to discover motion paths. Consequently, all processing must be performed in a single pass over the stream.

We propose a two-tier approach. The first tier involves a one-pass greedy algorithm, termed RayTrace, running on each object independently. The second is a discovery strategy, termed SinglePath, that runs on the coordinator and utilizes a lightweight index structure, termed MotionPath, for storing the hot motion paths. The RayTrace algorithm acts as a filter maintaining a permissible spatiotemporal extent, termed *Spatial Safe Area* (SSA), around the object's trajectory. When a location measurement falls outside the SSA, the current state of the object is sent to the coordinator; a response will arrive in the next epoch. The coordinator executes the discovery strategy in the following manner. It processes messages from all reporting objects and extracts motion paths for each of them using information found in MotionPath. Finally, in the upcoming epoch, it sends a message to each reporting object informing them about the motion path they just crossed. We present in detail the RayTrace algorithm in Section 5.5, while we discuss the index structures and discovery strategy in Section 5.6.

5.5 Filtering Position Updates

The RayTrace algorithm constructs a permissible spatiotemporal extent (the aforementioned SSA) around an object's trajectory, given some tolerance. RayTrace is a one-pass greedy algorithm that requires only constant per-measurement processing time and constant space. We first examine the case of tolerance ϵ; the adaptation to uncertainty, modeled by tolerance (ϵ, δ) is presented in Section 5.5.1.

The SSA is a spatiotemporal extent defined by the area between an initial timepoint $\langle \mathbf{s}, t_s \rangle$ and a rectangle, termed *Final Safe Area* (FSA), at time t_e. The main property of SSA is that a motion path $\mathbf{se}$ exists such that $\mathbf{e}$ lies inside FSA and

the object crosses it during $[t_s, t_e]$. The objective of the RayTrace algorithm is to identify the latest timestamp t_e, and hence the largest SSA, such that a motion path can be found for the $[t_s, t_e]$ interval. Once RayTrace determines that the SSA cannot grow larger without violating the tolerance parameters, it notifies the coordinator about its state. The coordinator executes a discovery strategy and responds with a timepoint $\langle \mathbf{e}, t_e \rangle$, which serves as the initial timepoint for the new SSA to be constructed by RayTrace. The requirement that the endpoint is the next initial timepoint guarantees that we construct a covering motion path set.

The SSA is uniquely identified by an initial timepoint $\langle \mathbf{s}, t_s \rangle$ and an FSA at time t_e. Alternatively, we can denote the SSA as a time parameterized rectangle $(\mathbf{l}(t), \mathbf{u}(t))$ for $t_s \leq t \leq t_e$, so that $\mathbf{l}(t_s) \equiv \mathbf{u}(t_s) \equiv \mathbf{s}$ and $(\mathbf{l}(t_e), \mathbf{u}(t_e))$ defines the FSA. We use the notation SSA$|t_i$ to imply the projection of the SSA at time t_i; thus, FSA = SSA$|t_e$.

Algorithm 1 illustrates RayTrace in detail. In the following we describe the most important step in RayTrace, updating the SSA. Given tolerance ϵ, each timepoint $\langle \mathbf{p}_i, t_i \rangle$ is associated with a square $\mathbf{Q}$ of side 2ϵ around $\mathbf{p}_i$, termed *tolerance square*. When examining a timepoint $\langle \mathbf{p}_i, t_i \rangle$, (Lines 24–40 in Algorithm 1), RayTrace must update the SSA so that its projection at t_i is not greater that the tolerance square. It first computes the projection SSA$|t_i$ (Lines 26–27):

$$\mathbf{l}(t_i) = \mathbf{l}(t_s) + \frac{t_i - t_s}{t_e - t_s}(\mathbf{l}(t_e) - \mathbf{l}(t_s))$$

$$\mathbf{u}(t_i) = \mathbf{l}(t_s) + \frac{t_i - t_s}{t_e - t_s}(\mathbf{u}(t_e) - \mathbf{l}(t_s)).$$

RayTrace also constructs the tolerance square $\mathbf{Q}$ (Lines 29–30). Then RayTrace examines if an intersection between SSA$|t_i$ and $\mathbf{Q}$ exists. If it does, then the SSA is updated by setting SSA$|t_i$ to be the intersection (Lines 33–34) and proceeds to process the next timepoint. If an intersection does not exist, the SSA cannot extend any further in time. RayTrace sends its state to the coordinator (Line 38) and goes into waiting mode (Line 36), expecting the server response. The state message $\langle \mathbf{l}(t_s), t_s, \mathbf{l}(t_e), \mathbf{u}(t_e), t_e \rangle$ includes the initial timestamp t_s, the start point $\mathbf{s} \equiv \mathbf{l}(t_s)$, the final timestamp t_e and the FSA $(\mathbf{l}(t_e), \mathbf{u}(t_e))$. As long as an object is in waiting mode, it stores incoming timepoints in a buffer (Lines 37 and 11). When the next epoch arrives, RayTrace receives the final timepoint that becomes the initial timepoint for a new SSA (Lines 13–16) and proceeds with processing new timepoints.

Example 5.5.1. *Figure 5.11 illustrates the process of updating the SSA, which is depicted in all figures as the shaded spatiotemporal extent. The initial timepoint is $\langle \mathbf{p}_0, t_0 \rangle$; assume that a new timepoint $\langle \mathbf{p}_1, t_1 \rangle$ arrives, which defines the tolerance square $\mathbf{Q}_1$. Since this is the first timepoint after the initial one, the SSA$|t_1$ becomes equal to $\mathbf{Q}_1$, as demonstrated in Figure 5.11(a). Next, $\langle \mathbf{p}_2, t_2 \rangle$ arrives defining the tolerance square $\mathbf{Q}_2$, illustrated in Figure 5.11(b). The projection of the SSA at the $t = t_2$ plane (SSA$|_{t_2}$) is then intersected with $\mathbf{Q}_2$. Finally, the result of the intersection forms the projection SSA$'|_{t_2}$ shown in Figure 5.11(c).*

The RayTrace algorithm requires only constant space to store the SSA information; a total of three points and two timestamps — i.e., the state of the object. The main task of the algorithm is to maintain and update the SSA. For each newly arriving timepoint, this process (projecting and intersecting) requires only constant

time. Also, assuming that a response from the coordinator comes in a timely manner, i.e., at the next epoch, the buffer does not grow indefinitely. Therefore, RayTrace requires $O(1)$ space and $O(1)$ time per processed timepoint.

5.5.1 Handling Uncertainty

We first consider the case of a single spatial dimension. A timepoint $\langle X_i, t_i \rangle$ in this case implies that the location X_i of the object at t_i is a random variable. Given tolerance ϵ, δ and assuming that X_i follows a normal distribution with known parameters, we show how to adapt the RayTrace algorithm. The objective is to define a tolerance interval for this timepoint.

The location sensing device reports the mean value x_i and the standard deviation σ_i of a measurement. We assume that the actual location follows a normal distribution, i.e., $X_i \sim N(x_i, \sigma_i^2)$. Let x_i' denote a location that is close to X_i. According to the definition of proximity in Section 5.4 we require:

$$Pr\left(|X_i - x_i'| \leq \epsilon\right) \geq 1 - \delta,$$

or equivalently:

$$Pr\left(X_i \in [x_i' - \epsilon, x_i' + \epsilon]\right) \geq 1 - \delta. \tag{5.1}$$

Thus, the probability that X_i is in the $[x_i' - \epsilon, x_i' + \epsilon]$ interval must be above $1 - \delta$. Figure 5.12 illustrates the probability density function (pdf) of X_i. Equation 5.1 states that the shaded part of the pdf has area more than $1 - \delta$. This area is calculated as:

$$\Phi\left(\frac{x_i' + \epsilon - x_i}{\sigma_i}\right) - \Phi\left(\frac{x_i' - \epsilon - x_i}{\sigma_i}\right),$$

using the standard cumulative distribution function $\Phi(z) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right)$. The error function values $\text{erf}(z)$ are typically precomputed and a table lookup is sufficient for estimating the area.

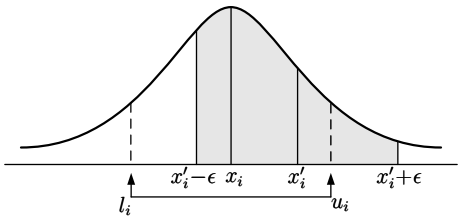


Figure 5.12: *Calculating tolerance square for* $\langle X_i, t_i \rangle$

As shown in Figure 5.12, x_i' should not be far from the mean value x_i; otherwise, the shaded area cannot be larger than δ. Let l_i (u_i) be the lowest (highest) value that x_i' can take without violating Equation 5.1, i.e., l_i, u_i are the solutions to the equation:

$$\Phi\left(\frac{x_i' + \epsilon - x_i}{\sigma_i}\right) - \Phi\left(\frac{x_i' - \epsilon - x_i}{\sigma_i}\right) = 1 - \delta \tag{5.2}$$

Equation 5.2 can be solved numerically in two ways: (i) perform a binary search on Φ's lookup table for those x_i' values satisfying the equation (exploiting Φ's monotonicity); (ii) precompute a lookup table which provides l_i, u_i given ϵ and δ and

Algorithm 1 RayTrace algorithm

Procedure RayTrace
Input: Timepoint Stream $\{\langle \mathbf{p}_i, t_i \rangle\}$
Input: Initial Timepoint $\langle \mathbf{p}_0, t_0 \rangle$
Input: Tolerance ϵ
$t_s \leftarrow t_0$; $t_e \leftarrow t_0$; // Initialization of SSA
$\mathbf{l}(t_s) \leftarrow \mathbf{p}_0$;
$waiting \leftarrow false$;
$buf \leftarrow \{\}$;
while 1 **do**
  Retrieve timepoint $\langle \mathbf{p}_k, t_k \rangle$;
  buf.pushBack($\langle \mathbf{p}_k, t_k \rangle$) ;
  if *waiting* and time is next epoch **then**
    Retrieve timepoint from coordinator $\langle \mathbf{p}_{coord}, t_{coord} \rangle$;
    $t_s \leftarrow t_{coord}$; $t_e \leftarrow t_{coord}$; // Reset SSA
    $\mathbf{l}(t_s) \leftarrow \mathbf{p}_{coord}$;
    $waiting \leftarrow false$
  end if
  while !$waiting$ and buf ! = {} **do**
    $\langle \mathbf{p}_i, t_i \rangle \leftarrow buf$.popFront() ;
    if $t_e = t_s$ **then** // This is the first timepoint after t_s
      $t_e \leftarrow t_i$;
      $\mathbf{l}(t_e) \leftarrow \mathbf{p}_i - (\epsilon, \epsilon)$;
      $\mathbf{u}(t_e) \leftarrow \mathbf{p}_i + (\epsilon, \epsilon)$;
    else
      // Calculate FSA = SSA$|t_i$ at time t_i
      $\mathbf{l}(t_i) \leftarrow \mathbf{l}(t_s) + \frac{t_i - t_s}{t_e - t_s}(\mathbf{l}(t_e) - \mathbf{l}(t_s))$;
      $\mathbf{u}(t_i) \leftarrow \mathbf{l}(t_s) + \frac{t_i - t_s}{t_e - t_s}(\mathbf{u}(t_e) - \mathbf{l}(t_s))$;
      // Calculate tolerance area $(\mathbf{l}_i, \mathbf{u}_i)$ around $\mathbf{p}_i$
      $\mathbf{l}_i \leftarrow \mathbf{p}_i - (\epsilon, \epsilon)$;
      $\mathbf{u}_i \leftarrow \mathbf{p}_i + (\epsilon, \epsilon)$;
      if intersects($(\mathbf{l}(t_i), \mathbf{u}(t_i)), (\mathbf{l}_i, \mathbf{u}_i)$) **then**
        $t_e \leftarrow t_i$; // Update SSA
        $\mathbf{l}(t_e) \leftarrow \max\{\mathbf{l}(t_i), \mathbf{l}_i\}$;
        $\mathbf{u}(t_e) \leftarrow \min\{\mathbf{u}(t_i), \mathbf{u}_i\}$;
      else // Send message to coordinator
        $waiting \leftarrow true$ // Go into waiting mode
        buf.pushBack($\langle \mathbf{p}_i, t_i \rangle$)
        Send state $\langle \mathbf{l}(t_s), t_s, \mathbf{l}(t_e), \mathbf{u}(t_e), t_e \rangle$
      end if
    end if
  end while
end while
End Procedure

simply perform a single lookup per instance. The latter option is the most efficient method requiring constant time per timepoint. Note that since lookup tables are given for the $N(0,1)$ distribution, a simple transformation is required for arbitrary mean and standard deviation.

Observe that $[l_i, u_i]$ serves as the tolerance interval for the timepoint $\langle X_i, t_i \rangle$ with mean value x_i and standard deviation σ_i. Consequently, the RayTrace algorithm can be straightforwardly adapted to construct an SSA given uncertainty in the input data.

An important point is that for given ϵ, δ a timepoint might have standard deviation σ_i such that Equation 5.2 has no solutions. To avoid this pitfall, a proactive approach would be to set more relaxed tolerance bounds, assuming knowledge of the typical imprecision in the location sensing devices. A retroactive approach would be to assign some predefined minimal tolerance area to these timepoints.

In the case of xy plane, the location $\mathbf{P}_i = (X_i, Y_i)$ of an object at time t_i is a random vector following a joint 2d normal distribution: $\mathbf{P}_i \sim N(\mathbf{p}_i, \mathbf{\Sigma}_i)$. A location (x'_i, y'_i) is close to (X_i, Y_i) if:

$$Pr\left(\max\{|X_i - x'_i|, |Y_i - y'_i|\} \leq \epsilon\right) \geq 1 - \delta,$$

or equivalently:

$$Pr\left((|X_i - x'_i| \leq \epsilon) \wedge (|Y_i - y'_i| \leq \epsilon)\right) \geq 1 - \delta.$$

Assuming independence among x, y measurements (hence, $\mathbf{\Sigma}_i = \text{diag}\left((\sigma_i^x)^2, (\sigma_i^y)^2\right)$) the last equation becomes:

$$Pr\left(|X_i - x'_i| \leq \epsilon\right) \cdot Pr\left(|Y_i - y'_i| \leq \epsilon\right) \geq 1 - \delta. \tag{5.3}$$

To simplify Equation 5.3, we require the failure probability to be less than $\frac{\delta}{2}$ for each dimension, i.e.:

$$Pr(X_i \in [x'_i - \epsilon, x'_i + \epsilon]) \geq 1 - \frac{\delta}{2},\ Pr(Y_i \in [y'_i - \epsilon, y'_i + \epsilon]) \geq 1 - \frac{\delta}{2},$$

since $(1 - \frac{\delta}{2})^2$ is marginally larger than $1 - \delta$ for small δ values. Therefore, using such a simplification, it is easy to revert to the single dimensional case and apply the methodology previously described.

5.6 Spatiotemporal Synopses

The coordinator performs three basic tasks: (i) stores detected motion paths, (ii) maintains their hotness, and (iii) executes a discovery strategy, processing the state of reporting objects. In the following, we discuss each task in detail.

5.6.1 Storing Motion Paths

We use a lightweight grid-based index to store motion paths. The entire space is partitioned into a predetermined number of cells and the endpoints of each motion path are indexed, rather than the linear shape of the path itself. Every cell contains a list of entries about endpoints that fall inside its area. Apart from storing coordinates

of these endpoints, each index entry also stores the respective motion path id and the coordinates of the other endpoint. The list is sorted by motion path id and organized in a hash table. This allows for fast insertions and deletions, requiring constant (on average) time.

5.6.2 Hotness Maintenance

Recall that the hotness h_i of a motion path $\mathbf{p}_i\mathbf{p}_{i+1}$ is expressed as the number of objects that have crossed it within a sliding window extending to the past W time units from current time. To maintain this count for each motion path, we use a hash table and an event queue. The hash table uses as key the motion path id i, and stores for each i the corresponding number of objects h_i. The event queue updates the hash table when the exit timestamp t_e^i of an object expires from window W.

Assume that we detect that an object has crossed motion path $\mathbf{p}_i\mathbf{p}_{i+1}$ with id i at $[t_s, t_e]$. First, we increase (by one) the counter for $\mathbf{p}_i\mathbf{p}_{i+1}$ in the hash table. The counter will have to be decreased at time $t_e + W$, since the corresponding interval will completely fall outside the window W. To efficiently capture these interval expirations, upon updating h_i, we en-heap tuple $\langle t_e + W, i\rangle$ into the event queue.

The queue is sorted on expiry time, and its head corresponds to the next expiring interval considering all motion paths and intervals in the system. When the current time reaches the expiry time at the head of the heap, then: (i) the top entry is de-heaped, (ii) the hotness of the corresponding motion path is decreased in the hash table, and (iii) if the hotness becomes 0, the motion path is deleted from the grid and the hash table.

Each counter lookup or update in the hash table takes expected constant time. Every en-heap or de-heap operation in the event queue costs time logarithmic to the number of its entries. Thus, the overall computational overhead is low. Regarding space requirements, both structures are relatively concise and can be maintained in main memory.

5.6.3 The SinglePath Strategy

The SinglePath strategy processes all state messages $\{\langle \mathbf{s}^i, t_s^i, \mathbf{l}^i, \mathbf{u}^i, t_e^i\rangle\}$ received from the reporting objects. Note that we use the superscript index i to refer to object with id i. The objective of SinglePath is to find the hottest motion path that starts from $\mathbf{s}^i$ and finishes somewhere inside the FSA $(\mathbf{l}^i, \mathbf{u}^i)$. The rationale behind this policy is to minimize the number of paths introduced by any single object, utilizing motion paths already discovered and crossed by other objects. In order to exploit existing motion paths, one should first probe the grid index and examine all paths intersecting this FSA, since these paths could be most relevant to the current motion of object i. However, depending on the distribution of objects and the actual pattern of their movement, it may occur that no motion path matches the current state of that object. We distinguish three cases regarding the information retrieved from MotionPath for each object i:

1. There are *available motion paths* starting from $\mathbf{s}_i$ and ending somewhere inside the FSA $(\mathbf{l}^i, \mathbf{u}^i)$.

2. There is no available motion path, but there exist *available vertices*, i.e., motion path endpoints that fall inside FSA $(\mathbf{l}^i, \mathbf{u}^i)$.

3. No available motion path or vertex is found.

Note that Case 1 simply involves updating the hotness for a motion path that will be chosen among the available ones. However, Cases 2 and 3 entail construction of a new motion path for the object at hand and this path must be stored.

SinglePath attempts to identify motion paths and to compute hotness collectively for all objects, in order to reuse existing motion paths (thus, increasing their hotness) and avoid introduction of multiple new ones. The strategy first handles all objects for which available motion paths were identified (Case 1). Afterwards, it takes care of the remaining objects that received no path at all (Cases 2 and 3). All cases follow a two-phase paradigm: *generation of candidates* (motion paths or vertices) and *selection of hottest candidate*. Algorithm 2 shows in detail all steps involved; in the following, we discuss the most critical operations.

Handling candidate motion paths. Throughout this step, a search for qualifying motion paths is performed for each object (Function *GetCandidatePaths* called in Line 5). Initially, a range query is evaluated against the grid index, specifying a rectangle $(\mathbf{l}^i, \mathbf{u}^i)$ for each object i (Line 42). We obtain motion paths $\mathbf{s}^i\mathbf{p}_j$ that intersect $(\mathbf{l}^i, \mathbf{u}^i)$. Their respective hotness values h_j are obtained after performing a single lookup in the hash table. Let $\mathcal{AP}_i = \{\langle \mathbf{s}^i\mathbf{p}_j, h_j \rangle\}$ denote the set of available motion paths retrieved for object i (Lines 43–46). Note that the hotness of each path in $\mathcal{AP}_i$ associated to the i-th object should increase by one (Line 44), implying the potential influence of object i on the significance of any of these motion paths (i.e., if eventually were chosen as hottest). Note that hotness values are only temporarily adjusted in $\mathcal{AP}_i$, leaving intact the contents of the hash table.

As soon as index probing is finished, a new set $\mathcal{CP}_i$ defines the candidate motion paths obtained for object i; hence, $\mathcal{CP}_i = \mathcal{AP}_i$ (Line 5). We stress that other objects may also accentuate hotness of paths in $\mathcal{CP}_i$, since the sets of available motion paths are not disjoint (Lines 13–15). This is reasonable, considering that potential selection of a specific motion path for an object $j \neq i$ could modify hotness ranking among candidate paths for i. Finally, provided that the set $\mathcal{CP}_i$ of candidate motion paths is non-empty, the selection phase simply involves choosing the hottest path for each object i among those collected in its $\mathcal{CP}_i$ (Lines 17–20).

Handling candidate vertices. Recall that this stage affects only objects for which no motion path has been identified during the previous step. For each object i, it provides a set of candidate end vertices for a new path that will have its starting vertex at $\mathbf{s}^i$. Our goal is to choose the hottest possible vertex as the endpoint of this newly discovered motion path for object i. Intuitively, selection of the hottest vertex increases the chances that object i crosses a hot motion path immediately afterwards. Such vertices can be obtained from existing motion paths, while hotness of a vertex is calculated summing the hotness of each incoming motion path (implying multiple segments converging to them).

Let $\mathcal{AV}_i = \{\langle \mathbf{p}_j, h_j \rangle\}$ denote the set of available vertices $\mathbf{p}_j$ and their hotness h_j for object i. The construction of $\mathcal{AV}_i$ is detailed in function *GetCandidateVertices*. Similarly to Case 1, this set is obtained with a range query against the grid (Line 51). The hotness of each vertex is calculated by summing up the hotness of all converging motion paths (Lines 54–56).

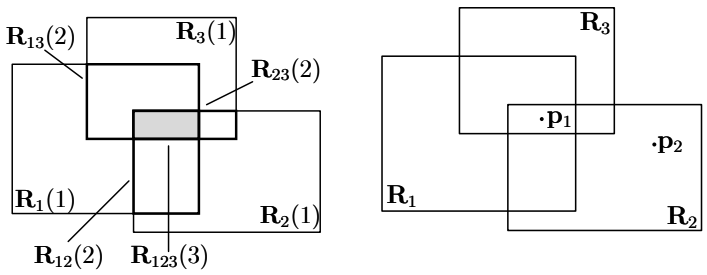
However, it is not sufficient to only consider the vertices of existing motion paths inside the FSA. Specifically:

Algorithm 2 SinglePath Strategy

1: **Procedure** *InsertMotionPaths*
2: **Input:** Object $States = \{\langle \mathbf{s}^i, t_s^i, \mathbf{l}^i, \mathbf{u}^i, t_e^i \rangle\}$
3: $\mathbf{R_{all}} \leftarrow \emptyset$; *//Memory-resident structure for FSA's*
4: **for** each state $\langle \mathbf{s}^i, t_s^i, \mathbf{l}^i, \mathbf{u}^i, t_e^i \rangle$ **do**
5: $\quad \mathcal{CP}_i \leftarrow GetCandidatePaths(\mathbf{s}^i, \mathbf{R}(\mathbf{l}^i, \mathbf{u}^i))$;
6: $\quad \mathbf{R_{all}} \leftarrow \mathbf{R_{all}} \cup \{\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)\}$;
7: **end for**
//Identify overlaps among final safe areas
8: **for all** $\mathbf{R_k} \in \mathbf{R_{all}}$ **do**
9: $\quad$ Calculate overlapping areas $\mathbf{R}_{\{j\}} = \bigcap_{k \in \{j\}} \mathbf{R}_k$
10: $\quad \mathbf{R}_{\{j\}}.count \leftarrow |\ \{j\}\ |$;
11: $\quad \mathbf{R_{all}} \leftarrow \mathbf{R_{all}} \cup \{\mathbf{R}_{\{j\}}\}$
12: **end for**
//Increase hotness of paths that appear in multiple $\mathcal{CP}$'s
13: **for** each motion path $mp_i \in \mathcal{CP}_i$ **do**
14: $\quad mp_i.hotness \leftarrow mp_i.hotness + |\ \{mp_i \in \mathcal{CP}_j, \forall j \neq i\}\ |$;
15: **end for**
//Selection phase
16: **for** each object i in *States* **do**
17: $\quad$ **if** $\mathcal{CP}_i \neq \emptyset$ **then**
18: $\quad\quad$ *//Case 1: Examine available motion paths*
19: $\quad\quad$ Choose motion path $mp_k \in \mathcal{CP}_i$ with *max hotness*
20: $\quad\quad$ Update hotness of mp_k at MotionPath index
21: $\quad$ **else**
22: $\quad\quad \mathcal{CV}_i \leftarrow GetCandidateVertices(\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i))$;
$\quad\quad$ *//Case 2: Check available end vertices of motion paths*
$\quad\quad$ *//Adjust vertex hotness according to potential overlaps*
23: $\quad\quad$ **for** each $\langle \mathbf{p}_j, h_j \rangle \in \mathcal{CV}_i$ **do**
24: $\quad\quad\quad$ Find smallest overlap $\mathbf{R}_k \in \mathbf{R_{all}}$ s.t. $\mathbf{p}_j \in \mathbf{R}_k$
25: $\quad\quad\quad h_j \leftarrow h_j + \mathbf{R}_k.count$;
26: $\quad\quad$ **end for**
$\quad\quad$ *//Case 3: Generate additional candidate vertices*
27: $\quad\quad h_m \leftarrow 0$;
28: $\quad\quad$ **for** each overlap $\mathbf{R}_k \in \mathbf{R_{all}}$ **do**
29: $\quad\quad\quad$ **if** $\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i) \cap \mathbf{R}_k \neq \emptyset$ **and** $\mathbf{R}_k.count > h_m$ **then**
30: $\quad\quad\quad\quad \mathbf{R}_m \leftarrow \mathbf{R}_k; h_m \leftarrow \mathbf{R}_k.count$;
31: $\quad\quad\quad$ **end if**
32: $\quad\quad$ **end for**
33: $\quad\quad v_m \leftarrow Centroid(\mathbf{R}_m)$;
34: $\quad\quad \mathcal{CV}_i \leftarrow \mathcal{CV}_i \cup \{\langle v_m, h_m \rangle\}$;
35: $\quad\quad$ Choose vertex $\mathbf{p}_k \in \mathcal{CV}_i$ with max hotness h_{max}
36: $\quad\quad$ Insert motion path $\langle \mathbf{s}^i \mathbf{p}_k, h_{max} \rangle$ at MotionPath index
37: $\quad$ **end if**
38: **end for**
39: **End Procedure**

40: **Function** *GetCandidatePaths*(vertex $\mathbf{s}^i$, rectangle $\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)$)
41: *Initialization:* $\mathcal{AP}_i \leftarrow \emptyset$
//Search MotionPath *index*
42: $\mathcal{P}_i \leftarrow$ motion paths $\{\mathbf{s}^i \mathbf{p}_j\}$ s.t. $\mathbf{p}_j \in \mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)$;
43: **for** each motion path $\mathbf{s}^i \mathbf{p}_j \in \mathcal{P}_i$ **do**
44: $\quad h_j \leftarrow hotness(\mathbf{s}^i \mathbf{p}_j) + 1$; *//Look-up in hash table*
45: $\quad \mathcal{AP}_i \leftarrow \mathcal{AP}_i \cup \{\langle \mathbf{s}^i \mathbf{p}_j, h_j \rangle\}$;
46: **end for**
47: **return** $\mathcal{AP}_i$
48: **End Function**

49: **Function** *GetCandidateVertices*(rectangle $\mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)$)
50: *Initialization:* $\mathcal{AV}_i \leftarrow \emptyset$;
//Search MotionPath *index*
51: $\mathcal{V}_i \leftarrow$ end vertices of motion paths s.t. $\mathbf{p}_j \in \mathbf{R}(\mathbf{l}^i, \mathbf{u}^i)$;
52: **for** each *distinct* vertex $\mathbf{p}_j \in \mathcal{V}_i$ **do**
53: $\quad h_j \leftarrow 0$;
$\quad$ *//Sum up hotness of all converging paths*
54: $\quad$ **for** each motion path $\mathbf{qp}_j$ terminating at $\mathbf{p}_j$ **do**
55: $\quad\quad h_j \leftarrow h_j + hotness(\mathbf{qp}_j)$;
56: $\quad$ **end for**
57: $\quad \mathcal{AV}_i \leftarrow \mathcal{AV}_i \cup \{\langle \mathbf{p}_j, h_j \rangle\}$;
58: **end for**
59: **return** $\mathcal{AV}_i$
60: **End Function**



(a) Intersections (b) Choosing a Vertex

Figure 5.13: *Considering overlapping rectangles for additional candidate vertices*

(i) $\mathcal{AV}_i$ may be empty if no motion path intersects the current FSA, so no vertices will be returned (Case 3). Therefore, a new vertex must be generated, in a way that takes into account motion patterns of other objects as well. This policy increases the chance that new vertices could also serve as endpoints of other motion paths in the future. Thus, we can avoid further segmentation of paths.

(ii) When calculating hotness for a vertex, we must also take into account the possibility that the same vertex may be returned for other objects as well, thus increasing the probability that this vertex might be more suitable for selection.

(iii) Newly generated motion paths for other objects will also provide additional vertices that should not be missed.

We successfully collect additional candidate vertices (besides those in $\mathcal{AV}_i$), by examining intersections of objects' FSA rectangles. We maintain a structure $\mathbf{R_{all}}$ that processes the final safe areas $\mathbf{R}_i$ $(= (\mathbf{l}^i, \mathbf{u}^i))$ of all considered objects (Line 6), and calculates their overlaps $\mathbf{R}_{\{j\}} = \bigcap_{k\in\{j\}} \mathbf{R}_k$ (Lines 8–12). Each rectangle in $\mathbf{R_{all}}$ is associated with a count (its perceived "hotness"), expressing the number of rectangles that it overlaps with, i.e., $c_{\{j\}} = |\{j\}|$ (Line 10). The intuition is that if we are forced to choose an arbitrary vertex, then its hotness should be as high as the count of the smallest stored rectangle in which it resides. This observation is better illustrated with the following example.

Example 5.6.1. *Consider three objects and their respective FSAs,* $\mathbf{R}_1$*,* $\mathbf{R}_2$ *and* $\mathbf{R}_3$*, that intersect with each other constructing intersections* $\mathbf{R}_{12}$*,* $\mathbf{R}_{23}$*,* $\mathbf{R}_{13}$ *and* $\mathbf{R}_{123}$*. Figure 5.13(a) illustrates all original FSA's and their overlaps, along with their counts. Now, assume that there are no available vertices for objects 1 and 3 and that there is a single available vertex* $\mathbf{p}_2$ *for object 2 with hotness 1. If we choose* $\mathbf{p}_2$ *as the endpoint for that object's motion path, its hotness will become 2 (one for the existing motion path plus one for the newly discovered path). However, had we chosen a vertex inside* $\mathbf{R}_{123}$*, say* $\mathbf{p}_1$ *in Figure 5.13(b), and used that as the endpoint for the motion path of all objects, its hotness would be 3. Obviously, we should consider introducing additional vertices from the overlapping areas with the highest counts.*

Once $\mathcal{AV}_i$ has been found, we construct $\mathcal{CV}_i = \{\langle \mathbf{p}_j, h_j \rangle\}$, the set of candidate vertices for each object i as follows. The candidate set is initialized to the set of

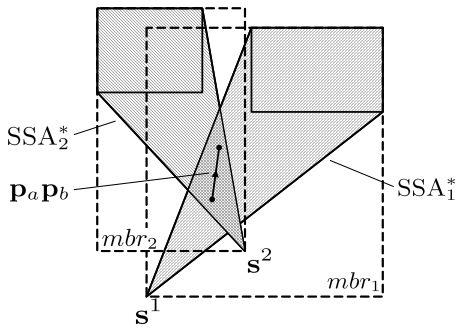


Figure 5.14: *Common motion path inside two SSA*s*

available vertices: $\mathcal{CV}_i = \mathcal{AV}_i$ (Line 22). Fix an object i and consider one of its available vertices $\mathbf{p}_j$ with hotness h_j. Let $\mathbf{R}_k$ be the smallest intersection in which $\mathbf{p}_j$ resides and let its associated count $c_k = \mathbf{R}_k.count$. Had we chosen vertex $\mathbf{p}_j$ as the endpoint of all objects whose FSA overlap with $\mathbf{R}_k$, then its hotness would become $h_j + c_k$. To reflect this potential influence, we increment by c_k the hotness of $\mathbf{p}_j$ in $\mathcal{CV}_i$ (Lines 23–26).

As soon as this update has been performed for all available vertices of all objects, we need to generate additional candidate vertices, as demonstrated in Example 5.6.1. In fact, we only need to generate a single additional vertex per object. Let $\mathbf{R}_m$ denote the intersected rectangle with the highest count c_m among those of object i, i.e., among those that fall inside FSA $\mathbf{R}_i$. Then, the newly generated candidate vertex for this object should lie inside $\mathbf{R}_m$ and, thus, must have hotness c_m. We choose one such vertex, e.g., by taking the centroid of $\mathbf{R}_m$, and insert it into $\mathcal{CV}_i$ (Lines 27–34). This scheme guarantees that candidate vertices exist even for objects that received neither a motion path nor a vertex from the index (Case 3). Finally, the selection phase per object i simply involves selecting the hottest candidate vertex among those in $\mathcal{CV}_i$ and inserting the newly created path into the index (Lines 35–36).

5.6.4 The MultiPath Strategy

The SinglePath insertion strategy tries to find the best motion path whose endpoint falls inside the final safe area per object. This, essentially, guarantees that the number of motion paths per object is minimized. The insertion strategy, termed MultiPath, we consider here, however, tries to generalize the movement of an object by considering polylines, instead of a single motion path, inside the SSA. We use the term *multipath* to denote such polylines. The rationale behind the MultiPath strategy is that the SSA (more accurately, the projection of SSA in the xy plane, denoted as SSA*) can contain important hot motion paths that we should consider as part of the trajectory synopsis. Consider Figure 5.14, where motion path $\mathbf{p}_a\mathbf{p}_b$ lies in the SSA*s of two reporting objects. We should consider this motion path as part of a candidate multipath of each object and increment its hotness by two.

Considering multipaths entails some difficulties. First, the reported *State* of each object carries no information about the timestamps found in its trajectory. Therefore, by definition (see Section 5.4), MultiPath can only extract weak trajectory synopses. Second, in continuation of the previous observation, for each motion path selected in a multipath, MultiPath should assign start and finish timestamps such

that the ϵ or (ϵ, δ) deficiency requirement is not violated. In other words, it is not sufficient that the multipath is contained in the projection SSA*; the multipath augmented with the timestamps should define a polyline in the *xyt* space that is completely contained inside the SSA. Third, the hotness of a multipath needs to be defined in order to choose the hottest one. The last point is the easiest to address: we simply define the hotness of a multipath as the average hotness of the motion paths it comprises.

In the initialization step, the MultiPath strategy retrieves all motion paths that are contained in the SSA* and all vertices that are contained in the FSA. To this end, a range query on the R-Tree is issued per object with range the MBR of the SSA*; Figure 5.14 illustrates in dashed lines the MBRs, mbr_1 and mbr_2, for the two SSA*s. We distinguish two categories for the reporting objects based on the information stored in MotionPath and returned by these range queries.

1. There are *available motion paths*, i.e., motion paths completely contained in the SSA*.

2. There are no available motion paths.

Objects in the latter category are processed using Step 2 of strategy SinglePath, i.e., we simply choose a single motion path starting in $\mathbf{s}^i$ and finishing inside the FSA. For all remaining objects i, let $\mathcal{AP}_i = \{\langle \mathbf{p}_j\mathbf{p}_{j+1}, h_j, q_j\rangle\}$ denote the non-empty set of available motion paths $\mathbf{p}_j\mathbf{p}_{j+1}$, their respective hotness h_j and their score q_j described in the following. Note that this is different definition than that of the SinglePath strategy.

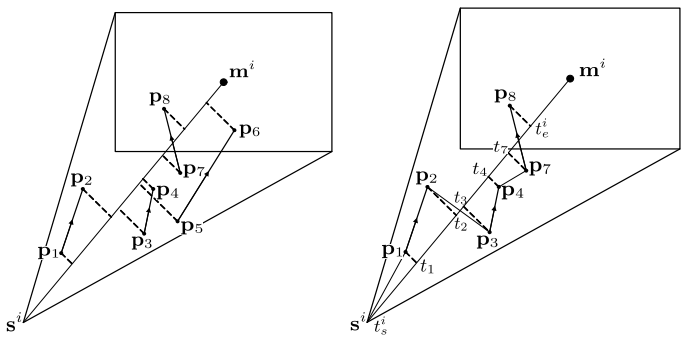
As a pre-processing step we execute the SinglePath strategy for all remaining objects and obtain the hottest motion path, $\mathbf{s}^i\mathbf{p}_e^i$, and its hotness h^i for each object i. This information is essential for the execution of MultiPath as we describe next.

Henceforth, MultiPath proceeds for each object i independently in four steps. The first step is to calculate the score q_j for each path in $\mathcal{AP}_i$. Then sets of paths, termed pathsets, are created and inserted in the set of pathsets $\mathcal{PS}_i$. In the third step, the candidate set of multipaths $\mathcal{CM}_i$ is created by augmenting pathsets from $\mathcal{PS}_i$. In the final step the hottest multipath is selected, appropriate timestamps are selected for each path and inserted into MotionPath. We describe these steps in detail.

An important notion is that of the *route segment*. Let $\mathbf{m}^i$ denote the midpoint of the FSA rectangle. The directed segment $\mathbf{s}^i\mathbf{m}^i$ is the route segment of object i, which serves two purposes: (i) it defines the general direction of movement, and (ii) it acts as a timeline for ordering the timestamps of paths by projecting them onto it. Figure 5.15(a) depicts the route segment for an SSA.

In Step 1, MultiPath assigns a score to each motion path $\mathbf{p}_j\mathbf{p}_{j+1}$ of $\mathcal{AP}_i$, defined as $q_j = h_j\left(1 + \cos\left(\mathbf{p}_j\mathbf{p}_{j+1}, \mathbf{s}^i\mathbf{m}^i\right)\right)$, where h_j is the hotness of the path and $\cos\left(\mathbf{p}_j\mathbf{p}_{j+1}, \mathbf{s}^i\mathbf{m}^i\right)$ is the cosine of the angle between the motion path and the route segment. Intuitively, the cosine awards motion paths that are aligned to the general direction of movement and penalizes those that follow the opposite direction. The entries in $\mathcal{AP}_i$ are ordered by their score.

In Step 2, MultiPath creates the set of pathsets, $\mathcal{PS}_i$, considering each motion path of $\mathcal{AP}_i$ in order. Each pathset is a set of motion paths; thus, $\mathcal{PS}_i = \{\{\mathbf{p}_j\mathbf{p}_{j+1}, \mathbf{p}_k\mathbf{p}_{k+1}, \ldots\}, \ldots\}$. The first pathset inserted is a singleton with the motion path of the highest score, $\{\mathbf{p}_1\mathbf{p}_2\}$. MultiPath then proceeds iteratively: let



(a) Example Set $\mathcal{AP}_i$ (b) Constructing a Multipath

Figure 5.15: *Example of the* MultiPath *insertion strategy*

$\{\mathbf{p}_1\mathbf{p}_2, \ldots, \mathbf{p}_j\mathbf{p}_{j+1}\}$ denote the last inserted pathset and let $\mathbf{p}_k\mathbf{p}_{k+1}$ be the next, in score order, motion path to be considered. If the projection of $\mathbf{p}_k\mathbf{p}_{k+1}$ on the route segment overlaps the projection of the pathset $\{\mathbf{p}_1\mathbf{p}_2, \ldots, \mathbf{p}_j\mathbf{p}_{j+1}\}$ then MultiPath skips $\mathbf{p}_k\mathbf{p}_{k+1}$. Otherwise, a new pathset, $\{\mathbf{p}_1\mathbf{p}_2, \ldots, \mathbf{p}_j\mathbf{p}_{j+1}, \mathbf{p}_k\mathbf{p}_{k+1}\}$, is inserted in the $\mathcal{PS}_i$. MultiPath then proceeds with the next in order motion path.

Example 5.6.2. *Consider the set of available motion paths* $\mathcal{AP}_i = \{\mathbf{p}_7\mathbf{p}_8,\ \mathbf{p}_3\mathbf{p}_4,\ \mathbf{p}_5\mathbf{p}_6,\ \mathbf{p}_1\mathbf{p}_2\}$ *sorted by their score and shown in Figure 5.15(a). We will create the set of pathsets* $\mathcal{PS}_i$*. The first entry will be the singleton pathset* $\{\mathbf{p}_7\mathbf{p}_8\}$*. The next motion path to consider is* $\mathbf{p}_3\mathbf{p}_4$*. Since its projection does not overlap the projection of the previous entered pathset a new entry,* $\{\mathbf{p}_7\mathbf{p}_8, \mathbf{p}_3\mathbf{p}_4\}$*, is inserted. Next, observe that the projection of the next motion path* $\mathbf{p}_5\mathbf{p}_6$ *overlaps with the projection of the previous pathset. Hence, we do not consider it further. Finally, the projection of the motion path* $\mathbf{p}_1\mathbf{p}_2$ *does not overlap with the previous pathset and, thus, a new pathset is inserted:* $\{\mathbf{p}_7\mathbf{p}_8, \mathbf{p}_3\mathbf{p}_4, \mathbf{p}_1\mathbf{p}_2\}$*. Therefore, the set of pathsets is* $\mathcal{PS}_i = \{\{\mathbf{p}_7\mathbf{p}_8\}, \{\mathbf{p}_7\mathbf{p}_8, \mathbf{p}_3\mathbf{p}_4\},\ \{\mathbf{p}_7\mathbf{p}_8, \mathbf{p}_3\mathbf{p}_4, \mathbf{p}_1\mathbf{p}_2\}\}$

In Step 3, MultiPath creates the candidate multipaths $\mathcal{CM}_i$ from the pathsets. The motion paths inside a pathset are arranged (ordered) according to how far from $\mathbf{s}^i$ their projection on the route segment is — note that by construction the projection defines a total order on motion paths. Fix a pathset of m motion paths and let $\{\mathbf{p}_1\mathbf{p}_2, \ldots, \mathbf{p}_m\mathbf{p}_{m+1}\}$ denote the arranged set of paths. The aim is to construct a multipath from s^i to the FSA by connecting the motion paths in pathset and inserting new paths as necessary. If vertex $\mathbf{p}_1$ is not $\mathbf{s}^i$ then $\mathbf{s}^i\mathbf{p}_1$ is added to the pathset. Similarly if $\mathbf{p}_{m+1}$ is not inside the FSA then $\mathbf{p}_{m+1}\mathbf{p}_e^i$ is added to the pathset, where $\mathbf{p}_e^i$ is the endpoint of the hottest motion path $\mathbf{s}^i\mathbf{p}_e^i$ returned by SinglePath. Finally, for every pair of consecutive paths $\mathbf{p}_j\mathbf{p}_{j+1}$, $\mathbf{p}_k\mathbf{p}_{k+1}$ for which $\mathbf{p}_{j+1}$ is not $\mathbf{p}_k$ the motion path $\mathbf{p}_{j+1}\mathbf{p}_k$ is added. The resulting pathset is a multipath and inserted into $\mathcal{CM}_i$ along with its average hotness.

Note that the set of candidate multipaths might not include the hottest motion path $\mathbf{s}^i\mathbf{p}_e^i$ returned by the pre-processing run of SinglePath. Therefore, we need to manually enter it, along with its hotness h^i, into $\mathcal{CM}_i$ for consideration by the next step.

In Step 4, the multipath of $\mathcal{CM}_i$ with the highest average hotness is selected. The final issue that remains is how to assign timestamps to the points of the hottest multipath. The route segment will aid this step acting as a timeline ranging from

t_s^i to t_e^i, as follows. The projection of the first vertex, $\mathbf{s}^i$, is assigned timestamp t_s^i and the projection of the last vertex, $\mathbf{e}$, is assigned timestamp t_e^i. The timestamps of all remaining vertices is calculated as follows. Consider $\mathbf{p}_j$ and its projection $\mathbf{p}'_j$; then, the timestamp assigned is $t_j = t_s^i + \frac{\mathbf{d}(\mathbf{s}^i,\mathbf{p}'_j)}{\mathbf{d}(\mathbf{s}^i,\mathbf{e})}(t_e^i - t_s^i)$, where $\mathbf{d}$ is the Euclidean distance.

Example 5.6.3. *Continuing Example 5.6.2, consider the last pathset entry of $\mathcal{PS}_i$ ordered according to the path projections onto the route segment: $\{\mathbf{p}_1\mathbf{p}_2, \mathbf{p}_3\mathbf{p}_4, \mathbf{p}_7\mathbf{p}_8\}$. In Step 3, the additional motion paths $\mathbf{s}^i\mathbf{p}_1$, $\mathbf{p}_2\mathbf{p}_3$, $\mathbf{p}_4\mathbf{p}_7$ are inserted to generate the candidate multipath $\mathbf{s}^i\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3\mathbf{p}_4\mathbf{p}_7\mathbf{p}_8$, shown in Figure 5.15(b). Assuming that this candidate is the hottest multipath, timestamps t_1, t_2, t_3, t_4, t_7 need to be calculated for vertices $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4, \mathbf{p}_7$ respectively, by projecting the vertices onto the route segment, as shown in Figure 5.15(b).*

5.7 Experiments

In this section we experimentally evaluate our methods for moving object synopses. In particular Section 5.7.1 focuses on spatial k-medoid synopses examining the proposed methods HBM, GBM, dHBM and dGBM. Then, Section 5.7.2 considers spatiotemporal motion path synopses and examines the proposed RayTrace algorithm and SinglePath strategy.

5.7.1 Spatial Synopses

We evaluate the performance of our methods, in terms of processing time (at the server), number of object updates (i.e., communication cost for the objects) and achieved average distance. We generate datasets of cardinality $|P|$ ranging between 10K and 200K objects as follows. For each tested $|P|$, we randomly select the initial position and the destination of each object among the points of a real spatial dataset (North America, available at *www.maproom.psu.edu/dcw*). The object follows a linear trajectory between the two points. Upon reaching the endpoint, a new random destination is selected and the process is repeated. At every timestamp, a percentage a of the objects move towards their endpoint (while the remaining ones remain static), covering a distance v. We refer to a and v as the object *agility* and *velocity*, respectively. The velocity is expressed as a percentage of the data space extent on the x axis (we have a $[0,10^4]\times[0,10^4]$ data space). The simulation length is 100 timestamps for each setting, and the reported measurements are the average observed values over all timestamps. We process continuous k-medoid queries for k between 2 and 512. We evaluate our four methods HBM, GBM, dHBM, and dGBM (where the latter two are the distributed versions of HBM and GBM). Also, we use as a competitor the TPAQ method with a main memory R-tree, since none of the other existing algorithms works for the large cardinalities tested, even for snapshot queries. To adapt TPAQ to medoid monitoring, we rerun it for the timestamps where (i) some of the medoids move, or (ii) the object updates affect the extents of the R-tree entries at the partitioning level. In each experiment we vary one parameter, while setting the remaining to their default values. The parameter ranges and defaults are shown in Table 5.2. For GBM and dGBM we fine-tuned the grid granularity (with respect to the average distance) for the default settings and use the best one

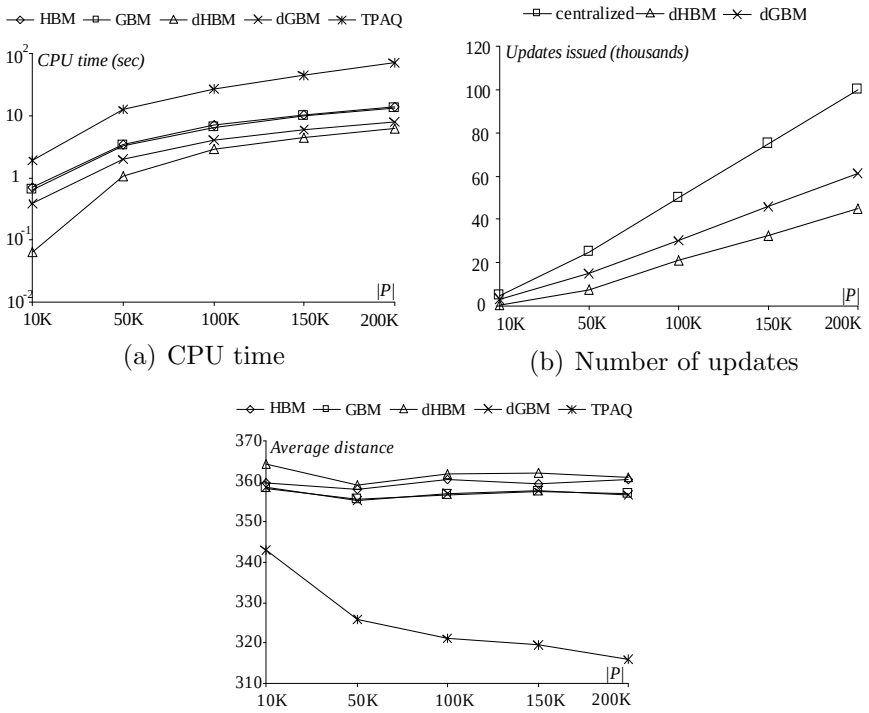
(100×100) in all our experiments. We use a machine with a 3.2 GHz Pentium IV CPU and 1 GB RAM.

Parameter	**Default**	**Range**
Dataset cardinality $\|P\|$	100K	10, 50, 100, 150, 200 (K)
No. of medoids k	32	2, 8, 32, 128, 512
Agility a	50%	10, 30, 50, 70, 100 (%)
Velocity v	0.5%	0.1, 0.3, 0.5, 0.7, 1 (%)
Leeway λ	300	100, 200, 300, 400, 500

Table 5.2: *Parameter ranges and default values*

In Figure 5.16, we measure the effect of object cardinality $|P|$, varying it from 10K to 200K objects and setting the other parameters to their defaults. Figure 5.16(a) shows the CPU cost (in logarithmic scale) for medoid maintenance per timestamp, i.e., the time to update the object index and the medoids. We observe that the centralized methods have similar cost (with GBM being slightly faster). The distributed algorithms have shorter running time, because they process fewer updates; dHBM (dGBM) takes less than 45% (60%) of the time of its centralized counterpart. dHBM is faster than dGBM, because the latter's safe regions are practically smaller, as they are bounded by the grid cell boundaries (leading to more reported updates and, thus, higher processing cost). Compared to our methods, TPAQ is slower by an order of magnitude, mainly due to the excessive update cost of its R-tree index. An important remark about Figure 5.16(a) (and all remaining CPU time charts) is that we focus on pure maintenance cost, i.e., we exclude the initial k-medoid computation. For the sake of completeness, the first-time medoid extraction for the default setting takes 12.9, 12.4 and 54.4 sec for HBM, GBM and TPAQ, respectively (the times for dHBM and dGBM are identical to HBM and GBM).

Figure 5.16(b) shows the number of updates sent to/processed by the server in the same experimental setup. All centralized methods (i.e., HBM, GBM, TPAQ) have the same communication cost, with the objects reporting their positions whenever they move. On the other hand, the safe regions of dHBM and dGBM save around 55% and 40% of these updates, respectively. dGBM avoids less updates than dHBM, due to the necessary updates required when the objects move to another cell, as explained in the context of Figure 5.16(a). Figure 5.16(c) illustrates the achieved distance for the various cardinalities, expressed in distance units in our $[0,10^4]\times[0,10^4]$ data space. We observe that the distributed methods compute only slightly worse medoid sets, verifying their efficacy. Note that both versions of GBM are better than those of HBM. The reason is that HBM is solely based on the one-dimensional Hilbert mapping, while GBM preserves a stronger connection to the original (two-dimensional) space, due to its spatial grid index. For a similar reason, TPAQ achieves 4 to 11% smaller distance than our methods, exploiting the graceful grouping properties of its R-tree. However, this benefit comes to a prohibitive update cost, leading to an excessive processing time (see Figure 5.16(a)). Another remark for TPAQ is that it improves with $|P|$; for a denser space, the nearest neighbor queries (in its final step) retrieve medoids that lie closer to the "ideal" geometric centroids of the k groups, leading to a lower distance.



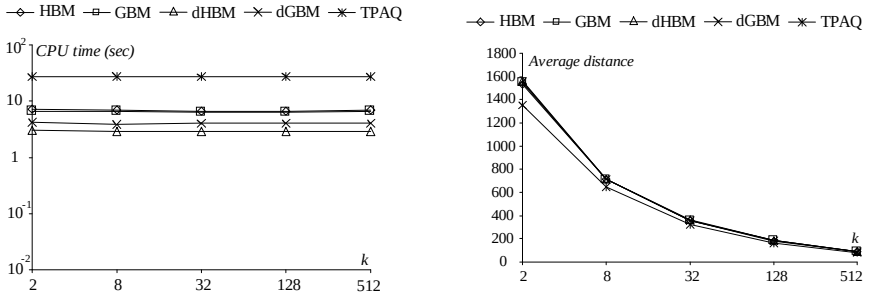
(a) CPU time

(b) Number of updates

(c) Average distance

Figure 5.16: *Performance versus dataset cardinality* $|P|$

In Figure 5.17 we use the default settings and vary k between 2 and 512. Figure 5.17(a) shows the CPU time. Again dGBM is the fastest, for the reasons explained above. We observe that the processing cost is almost constant for each method and unaffected by k. The reason is that, in all methods (and especially in TPAQ), the monitoring cost is dominated by the number of processed updates (mainly due to index maintenance), which is irrelevant to k. Furthermore, in our algorithms, for larger k, there are more medoids to maintain, but the offsets (to slide in the linear point order) are smaller. On the other hand, the average distance drops with k for all methods, and our techniques' difference from TPAQ decreases.



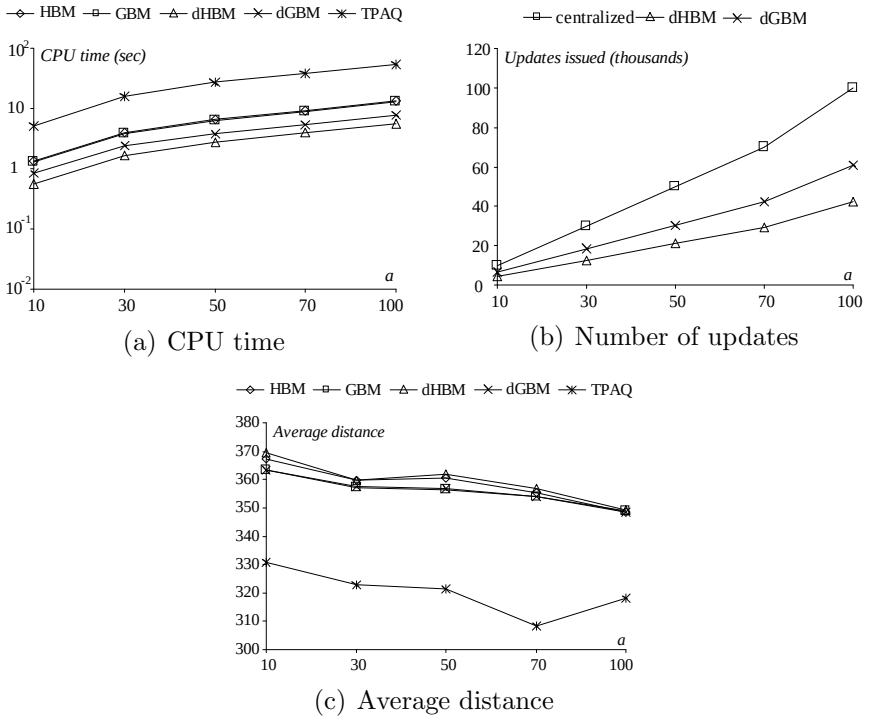
(a) CPU time

(b) Average distance

Figure 5.17: *Performance versus number of medoids* k

In Figure 5.18 we examine the effect of object agility a, with 10% up to 100% of the data points moving at each timestamp. The CPU cost (Figure 5.18(a)) increases with a due to the larger number of updates processed. Figure 5.18(b) shows the number of issued updates, which, as expected, is linear to a. In terms of average distance (Figure 5.18(c)), there is not much fluctuation; the small differences are

due to the randomness of the dataset generation.



(a) CPU time

(b) Number of updates

(c) Average distance

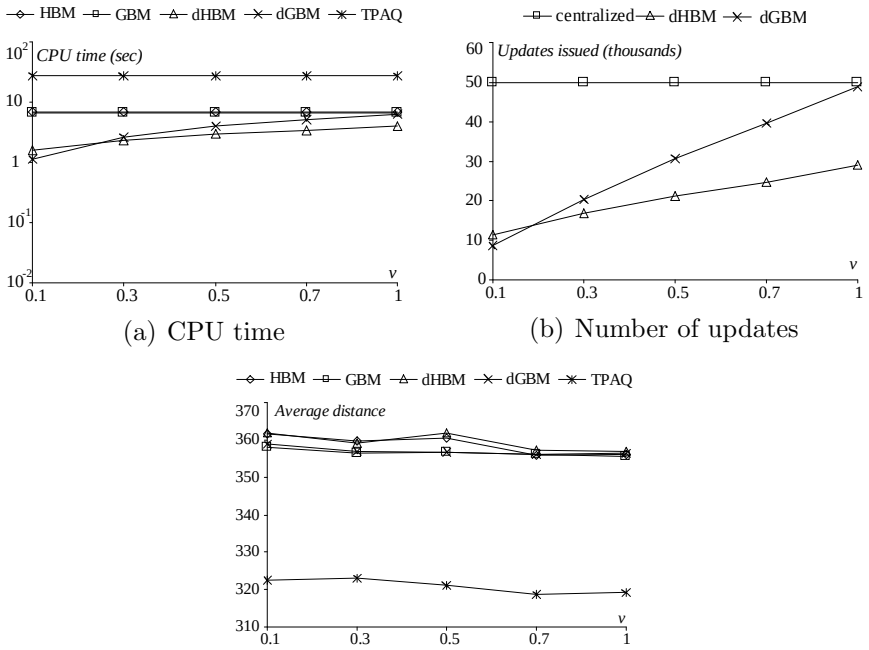
Figure 5.18: *Performance versus object agility a*

In Figure 5.19 we vary the object velocity v from 0.1 to 1% of the data space extent on the x dimension. Figure 5.19(a) shows the CPU time. The centralized methods are unaffected by v. On the other hand, the cost of the decentralized increases as more objects move outside their safe regions for larger v, sending more updates to the server for processing. This is also evident in Figure 5.19(b). Interestingly, for $v = 0.1\%$, dGBM incurs less object updates than dHBM (because its cells are large with respect to v, without practically limiting the safe regions), while for $v = 1\%$ their number is almost as high as for the centralized methods. The average distance (Figure 5.19(c)) is similar for all values of v.

Figure 5.20 investigates the effect of the leeway λ, varying it from 100 to 500. The performance of the centralized methods is identical, because they do not use safe regions. As shown in Figure 5.20(b), for $\lambda = 500$, dHBM achieves 65% reduction of the location updates. For dGBM, however, there is a marginal decrease, because the safe regions are restricted by the grid cells, rather than by λ. The number of updates has a direct impact on the CPU time and, thus, the trends in Figure 5.20(a) are similar as in Figure 5.20(b). In terms of average distance, λ affects only dHBM, whose performance deteriorates for larger λ. This trend verifies the tradeoff between update cost and medoid quality. On the other hand, dGBM is not affected because the server processes a similar set of updates.

5.7.2 Spatiotemporal Synopses

We empirically evaluate the performance of our framework, focusing on the RayTrace algorithm and the SinglePath strategy. To better gauge its effectiveness, we compare it against a Douglas-Peucker [26] variant, termed DP, that discovers spatial line segments close to the objects' movement. We must note that due to its nature, the



(a) CPU time

(b) Number of updates

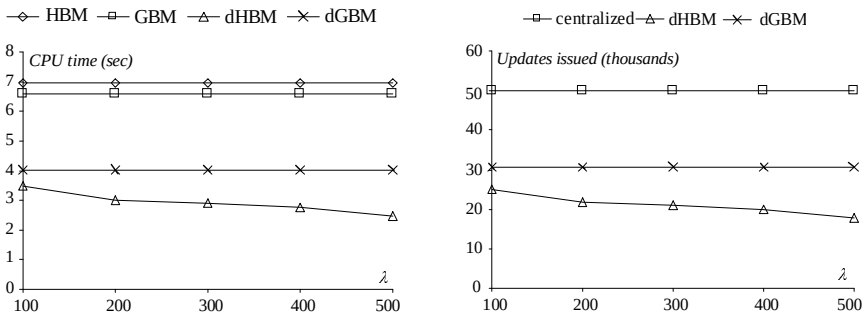
(c) Average distance

Figure 5.19: *Performance versus object velocity v*

output line segments do not constitute proper motion paths because they are disconnected and, in practice, they are hardly interpretable. Hence, DP is not directly comparable to our approach. Rather, as it purposefully benefits already existing line segments and is not bound by the strict covering motion path set requirements, DP is expected to assign higher hotness to segments when compared to our methodology.

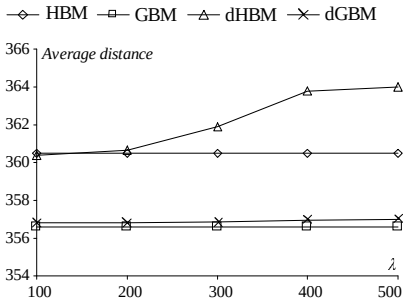
The DP Method. As described in Section 5.1, the windowed variations of Douglas-Peucker algorithm proposed in [66] offer concise trajectory synopses per object. However, unless objects follow exactly the same trajectory, all motion paths extracted will not have hotness greater than one. We choose to relax our requirements. In particular we allow selection of line segments that are close to objects' movements and ignore the time dimension. In this manner, we expect segments to achieve hotness that upper bounds the hotness achieved by motion paths. Whenever a new segment should be created between a starting point and the chosen floating point, we do not store it at once. Instead, we check whether an existing segment (produced earlier by another object) falls completely within the minimum bounding box (MBB) of the candidate segment. Each MBB is expanded by the tolerance value, to cope with uncertainty in objects' locations. In case such a segment exists, we need not store the candidate segment, but we must increase the hotness of the existing path. Otherwise, the new segment is stored with hotness 1. This simple policy can provide an even more dense approximation for each trajectory, with the additional benefit that many segments now belong to multiple object traces. On the other hand, connectivity between successive motion paths for each object is no longer preserved. Note that we measure DP's quality for the sake of comparison with SinglePath and exclude all time measurements. As observed in our evaluation, DP runs significantly faster than SinglePath because it simply performs one range query per discovered segment.

Experimental Setting. All algorithms were implemented in C++ and compiled with gcc on a 3GHz Intel Core 2 Duo CPU. All processing takes place in main



(a) CPU time

(b) Number of updates



(c) Average distance

Figure 5.20: *Performance versus leeway* λ

memory.

We generated synthetic datasets for trajectories of moving objects traveling on the main road network of greater Athens that covers an area of 250 km^2. We utilized a simplified graph of the network, assuming that nodes (representing major crossroads) are connected via straight linear links and not curved polylines (as in the real network). This network is illustrated in Figure 5.21 and consists of 1831 links connecting 1125 nodes in total. Links are ranked with weights, reflecting their significance in vehicle circulation. Thus, links are classified into four categories: motorways, highways, primary roads, and secondary roads.

Each object is initially assigned at a randomly chosen node. Whenever it is allowed to move, this object chooses to follow one of the outgoing links of that node. To make this decision, we calculate a ratio that expresses the relative weight of each such link compared to the total weight of all links connected to the current node. Finally, we randomly choose to follow a link with probability equal to its ratio. We assume that all objects have equal-length displacement s between successive positions, so that the next location will be along that link or at the opposite end node (at most). In the sequel, as long as an object does not cross a node, it continues its course along that link. Note, though, that movement is also controlled by another parameter that refers to the agility of moving objects. This means that, at each timestamp, only a portion α of the total number N of objects is allowed to move (decided randomly), while the rest remain stopped. Therefore, the inter-arrival time between positional measurements is not fixed for each object, but it fluctuates with time. As in a real traffic scenario, objects tend to follow main roads for large parts of their movement and enter into minor roads less frequently. To capture uncertainty, white noise is then added to object locations. In particular, a value randomly chosen between $-err$ and err is added to both coordinates. Although the data were generated by considering a fixed road network, the algorithms have

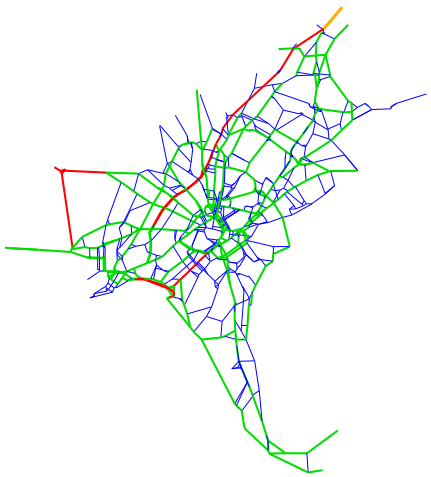


Figure 5.21: *Athens road network links.*

no knowledge of this fact, and, hence, cannot take advantage of it when discovering motion paths. Intuitively, we expect the algorithms to identify the most frequently traveled parts of the Athens' network.

We run a set of experiments for different parameter values. We consider N = 10,000, 20,000 and 100,000 objects that travel with fixed agility $a = 0.1$. During a timestamp, objects move $s = 10$ meters and take a location measurement with positional error $err = 1$ meter. We model uncertainty with ϵ tolerance[5]; we vary its value from $\epsilon = 1$ to 20 meters. Window size is fixed to $W = 100$ timestamps and at any timestamp we wish to recover the $k = 10$ hottest motion paths. In each experiment we vary a single parameter, while we set the remaining to their default values. The duration of every simulation is 250 timestamps and an epoch corresponds to 10 timestamps. Table 5.3 summarizes the parameters involved and their ranges; the default values are shown in bold.

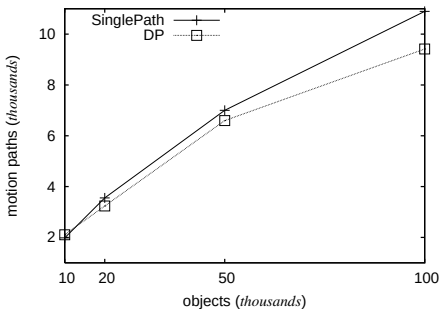
Parameter	**Range**
N	10000, **20000**, 100000 objects
Tolerance (ϵ)	1, 2, **10**, 20 meters
Positional error (err)	1 meter
Agility (α)	0.1
Displacement (s)	10 meters
Window size (W)	100 timestamps
k	10

Table 5.3: *Experimental parameters.*

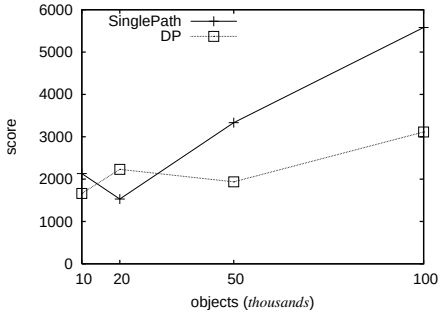
To measure the efficiency and quality of the SinglePath strategy we use three metrics. First, we measure the size of the index in terms of motion paths. Second, we calculate the score (see Section 5.4) of the top-k hottest motion paths discovered. Finally, we measure the processing time spent by the coordinator executing the SinglePath strategy. The reported/plotted measurements for the aforementioned performance factors correspond to average values per epoch.

Experimental Results. In the first set of experiments we vary the number of

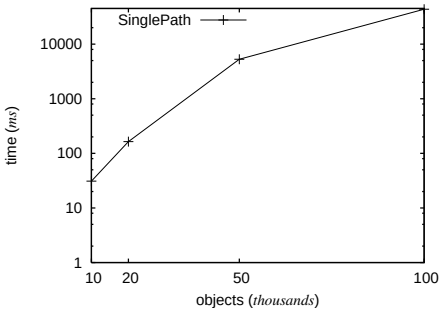
[5]We do not consider ϵ, δ tolerance since the processing involved is similar, as shown in Section 5.5.1



(a) Index Size vs N

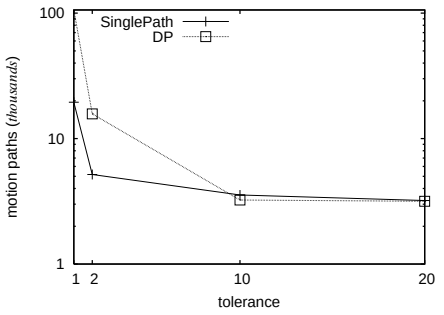


(b) Top-k Score vs N

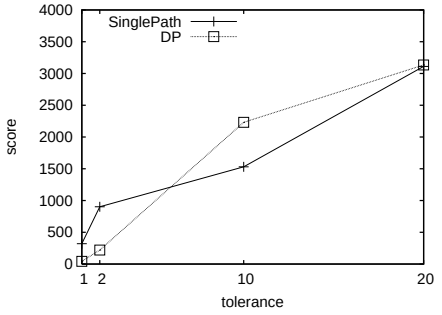


(c) Processing Time vs N

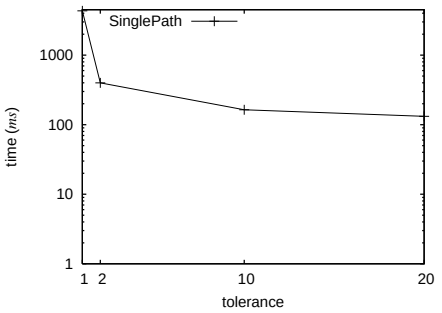
Figure 5.22: *Varying the number of objects*



(a) Index Size vs ϵ



(b) Top-k Score vs ϵ



(c) Processing Time vs ϵ

Figure 5.23: *Varying the tolerance parameter*

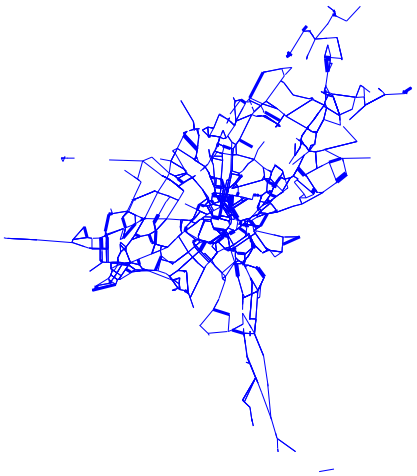


Figure 5.24: *The network as discovered by* SinglePath

objects from $N = 10,000$ to $100,000$ while the tolerance is fixed to $\epsilon = 10$ and show the results in Figure 5.22. Regarding the number of segments measured by the index size, Figure 5.22(a) clearly illustrates that DP inserts fewer segments. This is expected as DP enjoys more freedom and is not restricted to finding motion paths. SinglePath, on the other hand, must strictly identify motion paths that fit to some object's movement for a time interval. Note that even for 100,000 objects, SinglePath identifies only 16% more segments compared to DP, i.e., 10,896 versus 9,416.

Figure 5.22(b) shows the score for the top-10 hottest motion paths returned by the two methods for varying N values. In general, since DP identifies fewer total segments, their average hotness is larger than that of the motion paths found by SinglePath. Interestingly, for $N = 20,000$ SinglePath achieves higher score that DP. This is attributed to the fact that in this setting SinglePath extracts longer motion paths.

Figure 5.22(c) measures the average per epoch processing time spent by the coordinator running SinglePath. This running time essentially determines what the smallest epoch can be, since all processing must have finished by the next epoch so that objects exit the waiting mode of the RayTrace algorithm as soon as possible. As shown in the figure, for a large number of objects $N > 100,000$, processing time becomes close to 40 secs. To compensate for this behavior, one can choose to increase the tolerance parameter. As discussed in the following, higher ϵ values lead to reduced processing times.

Figure 5.23 measures the same metrics as before but fixes the number of objects to $N = 20,000$ and varies the tolerance parameter from $\epsilon = 1$ to 20. Figures 5.23(a) and 5.23(b) show that RayTrace significantly outperforms the benchmark, i.e., it discovers fewer motion paths that are both hotter and longer. When the tolerance increases, recall that the MBBs of the range queries that DP issues also increase. This results in more freedom when selecting segments. Regarding the scalability of processing time as ϵ increases, Figure 5.23(c) clearly illustrates the benefits of relaxing tolerance values. The processing time decreases by a factor greater than 3 when the ϵ increases from 2 to 20.

To better illustrate the effectiveness of our methods, Figure 5.24 draws the entire set of motion paths that have hotness greater than 0 within the time window. Comparing to the entire network shown in Figure 5.21, the SinglePath strategy man-

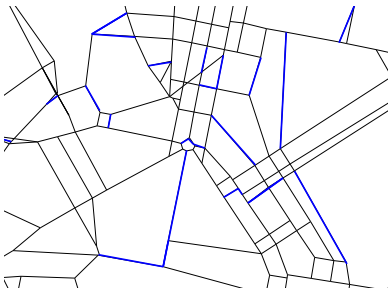


Figure 5.25: *Top 20 hottest motion paths in the center of Athens*

ages to accurately extract a set of motion paths that resembles the (unknown to SinglePath) network. Notice that motion paths with larger hotness are drawn with thicker lines. For completeness, Figure 5.25 focuses on the center of Athens and draws the top 20 hottest motion paths stored in the index.

5.8 Summary

In this chapter we focused on synopses for moving objects. We first studied the problem of dynamic maintenance for spatial k-medoid synopses. We considered a central server that continuously receives the locations of frequently moving objects and incrementally maintains their medoid set. Without making any assumption about the data moving patterns, our methods achieve low running times while keeping the medoid quality high. Furthermore, we considered distributed environments, where the data objects have limited power resources and attempt to preserve them by reducing the number of updates they transmit to the server. In this context, the server assigns safe regions to the objects, which report their position only when they exit their region. We evaluated our methods through extensive experiments and investigated tradeoffs between communication cost and spatial synopsis quality.

Then, we turned our attention to the temporal dimension and we investigate spatiotemporal motion path synopses. In particular, we proposed a framework for on-line maintenance of hot motion paths in order to detect frequently traveled trails of numerous moving objects. We considered a distributed setting, with a coordinator that maintains hotness and geometries of these paths in a spatiotemporal index, and many moving clients that issue updates only for important changes in their positions. We focused on motion patterns during the recent past, thus discarding obsolete paths that expire from a sliding time window. We assumed freely moving objects, i.e., not restricted by some network, and our techniques took into consideration uncertainty inherent in location readings while providing discrepancy guarantees for the discovered motion paths. Empirical simulations demonstrated the ability of our methodology to provide a dense representation of objects' movement, as well as its efficiency with respect to on-line maintenance of spatiotemporal synopses.

Chapter 6

Conclusions and Future Work

This thesis presented various methods for managing data streams using synopses. We focused on three data stream types, time series, update and moving objects streams. For the first two types we have devised general-purpose structures such as *wavelet synopses* and *sketches* for summarizing them. Then, we turned our attention to streams produced by objects moving freely in space. We presented techniques for creating and maintaining two distinct synopses, *spatial* and *spatiotemporal*.

6.1 Summary

Initially, we considered conventional wavelet synopses for generic multidimensional data streams. For time series streams we introduced two novel operators, *SHIFT-SPLIT*, operating directly on summaries of wavelet transformed data, allowing the management of streams so as an appropriate balance between the necessary space and time consuming is found. We analyzed costs for both the single dimensional case and the two forms of multidimensional transformation. There is a significant number of applications that can benefit from these operations. We have revisited some data maintenance scenarios, such as transforming massive multidimensional datasets and reconstructing large ranges from wavelet decomposed data, and utilized the SHIFT-SPLIT operations to draw comparisons with current state of the art techniques. Furthermore, we have provided solutions to some previously un-explored maintenance scenarios, namely, appending data to an existing transformation and approximation of multidimensional data streams. We demonstrated the effectiveness of the proposed techniques both analytically and experimentally, and we conjecture that the introduced operations can prove useful in a plethora of other applications, as the SHIFT-SPLIT operations stem from the general properties and behavior of wavelets.

For update multidimensional data streams, we have proposed the first known streaming algorithms for space- and time-efficient tracking of approximate wavelet summaries for both single and multidimensional data. Our approach relies on a novel, Group-Count Sketch (GCS) synopsis that, unlike earlier work, satisfies all three key requirements of effective streaming algorithms, namely polylogarithmic space usage, small, logarithmic update times (essentially touching only a small fraction of the GCS for each streaming update) and, polylogarithmic query times for computing the top wavelet coefficients from the GCS. Our experimental results with both synthetic and real-life data have verified the effectiveness of our approach,

demonstrating the ability of GCSs to support very high speed data sources.

Then, we introduced a novel indexing method for wavelet synopses, termed *Hierarchically Compressed Wavelet Synopses*. Our scheme seeks to improve the storage utilization of the wavelet coefficients and, thus, achieve improved accuracy to user queries by reducing the storage overhead of their coordinates. To accomplish this goal, our techniques exploit the hierarchical dependencies among wavelet coefficients that often arise in real datasets due to the existence of large spikes among neighboring data values and, more importantly, incorporate this goal in the synopsis construction process. We initially presented a dynamic programming algorithm, along with a streaming version of this algorithm, for constructing an optimal HCWS that minimizes the sum squared error given a space budget. We demonstrated that while in the worst case the benefit of our DP solution is only equal to the benefit of the conventional thresholding approach, it can often be significantly larger, thus achieving significantly reduced errors in the data reconstruction. We then presented an approximation algorithm with tunable guarantees leveraging a trade-off between synopsis accuracy and running time. Finally, we presented a fast greedy algorithm, along with a streaming version of this algorithm. We demonstrated that both of our greedy heuristics always exhibited near-optimal results in our experimental evaluation, with a running time on par with conventional thresholding algorithms. Extensions for multidimensional datasets, running time improvements for massive datasets and generalization to other error metrics were also introduced. Extensive experimental results demonstrate the effectiveness of HCWS against conventional synopsis techniques.

We studied the problem of dynamic maintenance for spatial k-medoid synopses. We considered a central server that continuously receives the locations of frequently moving objects and incrementally maintains their medoid set. Without making any assumption about the data moving patterns, our methods achieve low running times while keeping the medoid quality high. Furthermore, we considered distributed environments, where the data objects have limited power resources and attempt to preserve them by reducing the number of updates they transmit to the server. In this context, the server assigns safe regions to the objects, which report their position only when they exit their region. We evaluated our methods through extensive experiments and investigated tradeoffs between communication cost and spatial synopsis quality.

Finally, we turned our attention to the temporal dimension and investigated spatiotemporal motion path synopses. In particular, we proposed a framework for on-line maintenance of hot motion paths in order to detect frequently traveled trails of numerous moving objects. We considered a distributed setting, with a coordinator that maintains hotness and geometries of these paths in a spatiotemporal index, and many moving clients that issue updates only for important changes in their positions. We focused on motion patterns during the recent past, thus discarding obsolete paths that expire from a sliding time window. We assumed freely moving objects, i.e., not restricted by some network, and our techniques took into consideration uncertainty inherent in location readings while providing discrepancy guarantees for the discovered motion paths. Empirical simulations demonstrated the ability of our methodology to provide a dense representation of objects' movement, as well as its efficiency with respect to on-line maintenance of spatiotemporal synopses.

6.2 Future Work

During the course of this dissertation, we have identified the following interesting aspects that we propose as future work.

- Recently, there has been a lot of work regarding wavelet synopses over time series streams when the measured error is different than SSE. Although many methods have been proposed, all of them are plagued with high computational and storage costs. A nice alternative would be to design heuristic algorithms that approximately produce good synopses, under any error metric, with low memory and space requirements.

- Another direction for wavelet synopses is the design of algorithms that construct synopses suitable for range-aggregate queries. With only a few exceptions, the relevant literature is concerned about approximating individual values and not ranges of values. The problem of optimizing for a broader domain of queries is still open.

- The Group Count Sketch (GCS) presented in this thesis is a powerful summarization structure for update streams. We have demonstrated its usefulness by constructing wavelet synopses. A relevant interesting topic would be the application of GCS in maintaining multidimensional histograms, another popular summarization technique.

- Regarding spatial synopses for moving objects streams, the methods proposed in this thesis for construction of k-medoids offer no quality guarantees. An interesting direction would be to consider synopses that offer a tunable error in approximating the objects' current positions.

- There is little bibliography on spatiotemporal synopses. A challenging extension of the work presented in this thesis would be to provide a fully decentralized algorithm for extracting hot motion path synopses, i.e., without the need for a coordinator. This would require the motion path information indexed in the coordinator to be distributed among the objects themselves. Several issues regarding index content handling and handing-over arise.

In conclusion, we believe that there is a plethora of interesting and novel topics relevant to stream synopses that need to be addressed. Hopefully this thesis will be an instigation for further research in this area.

Bibliography

[1] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 10–20, 1999.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 20–29, 1996.

[3] A. Anagnostopoulos, M. Vlachos, M. Hadjieleftheriou, E. J. Keogh, and P. S. Yu. Global distance-based segmentation of trajectories. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 34–43, 2006.

[4] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 49–60, 1999.

[5] S. Arora, P. Raghavan, and S. Rao. Approximation schemes for euclidean k-medians and related problems. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 106–113, 1998.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[7] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 28–39, 2003.

[8] R. Baraniuk and D. Jones. A signal-dependent time-frequency representation: fast algorithm for optimal kernel design. *ISP*, 42(1):134–146, 1994.

[9] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. Uldbs: Databases with uncertainty and lineage. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 953–964, 2006.

[10] A. Bulut and A. K. Singh. Swat: Hierarchical stream summarization in large networks. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 303–314, 2003.

[11] Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Proceedings of the International Conference on Mobile Data Management(MDM)*, pages 27–38, 2004.

[12] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The International Journal on Very Large Data Bases (VLDBJ)*, 15(3):211–228, 2006.

[13] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 111–122, 2000.

[14] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 693–703, 2002.

[15] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 491–502, 2005.

[16] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[17] G. Cormode, M. Garofalakis, and D. Sacharidis. Fast approximate wavelet tracking on streams. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 4–22, 2006.

[18] G. Cormode, M. N. Garofalakis, S. Muthukrishnan, and R. Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 25–36, 2005.

[19] G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 296–306, 2003.

[20] G. Cormode and S. Muthukrishnan. What's new: Finding significant differences in network data streams. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2004.

[21] A. Deligiannakis, M. N. Garofalakis, and N. Roussopoulos. A fast approximation scheme for probabilistic wavelet synopses. In *SSDBM*, pages 243–252, 2005.

[22] A. Deligiannakis, M. N. Garofalakis, and N. Roussopoulos. Extended wavelets for multiple measures. *ACM Transactions on on Database Systems (TODS)*, 32(2):10, 2007.

[23] A. Deligiannakis and N. Roussopoulos. Extended wavelets for multiple measures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 229–240, 2003.

[24] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, pages 588–599, 2004.

[25] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 61–72, 2002.

[26] D. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitised line or its caricature. *The Canadian Cartographer Journal*, 10(2):112–122, 1973.

[27] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 226–231, 1996.

[28] M. Ester, H.-P. Kriegel, and X. Xu. A database interface for clustering in large spatial databases. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 94–99, 1995.

[29] M. Ester, H.-P. Kriegel, and X. Xu. Knowledge discovery in large spatial databases: Focusing techniques for efficient class identification. In *Proceedings of the International Symposium on Advances in Spatial Databases (SSD)*, pages 67–82, 1995.

[30] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 63–72, 1999.

[31] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[32] M. N. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 476–487, 2002.

[33] M. N. Garofalakis and A. Kumar. Deterministic wavelet thresholding for maximum-error metrics. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 166–176, 2004.

[34] M. N. Garofalakis and A. Kumar. Wavelet synopses for general error metrics. *ACM Transactions on on Database Systems (TODS)*, 30(4):888–928, 2005.

[35] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 67–87, 2004.

[36] B. Gedik and L. Liu. Mobieyes: A distributed location monitoring service using moving location queries. *IEEE Transactions on Mobile Computing*, 5(10):1384–1402, 2006.

[37] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 389–398, 2002.

[38] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 79–88, 2001.

[39] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 454–465, 2002.

[40] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. One-pass wavelet decompositions of data streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(3):541–554, 2003.

[41] S. Guha. Space efficiency in synopsis construction algorithms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 409–420, 2005.

[42] S. Guha and B. Harb. Wavelet synopsis for data streams: minimizing non-euclidean error. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 88–97, 2005.

[43] S. Guha and B. Harb. Approximation algorithms for wavelet transform coding of data streams. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 698–707, 2006.

[44] S. Guha, C. Kim, and K. Shim. Xwave: Approximate extended wavelets for streaming data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 288–299, 2004.

[45] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(3):515–528, 2003.

[46] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 73–84, 1998.

[47] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *The International Journal on Very Large Data Bases (VLDBJ)*, 15(2):143–164, 2006.

[48] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 479–490, 2005.

[49] H. Imai and M. Iri. An optimal algorithm for approximating a piecewise linear function. *Journal of Information Processing*, 9(3):169–162, 1986.

[50] M. Jahangiri, D. Sacharidis, and C. Shahabi. SHIFT-SPLIT: I/O efficient maintenance of wavelet-transformed multidimensional data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 275–286, 2005.

[51] B. Jawerth and W. Sweldens. An Overview of Wavelet Based Multiresolution Analyses. *SIAM Review*, 36(3):377–412, 1994.

[52] C. S. Jensen, D. Lin, and B. C. Ooi. Continuous clustering of moving objects. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(9):1161–1174, 2007.

[53] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *Proceedings of the International Symposium on Spatial and Temporal Databases (SSTD)*, pages 364–381, 2005.

[54] P. Karras and N. Mamoulis. One-pass wavelet synopses for maximum-error metrics. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 421–432, 2005.

[55] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data – An Introduction to Cluster Analysis*. John Wiley & Sons, 1990.

[56] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Z. 0003. Approximate nn queries on streams with guaranteed error/performance bounds. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 804–815, 2004.

[57] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: A partition-and-group framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 593–604, 2007.

[58] X. Li, J. Han, J.-G. Lee, and H. Gonzalez. Traffic density-based discovery of hot routes in road networks. In *Proceedings of the International Symposium on Spatial and Temporal Databases (SSTD)*, pages 441–459, 2007.

[59] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 617–622, 2004.

[60] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 236–245, 2004.

[61] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 346–357, 2002.

[62] Y. Matias and D. Urieli. Optimal workload-based weighted wavelet synopses. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 368–382, 2005.

[63] Y. Matias and D. Urieli. Inner-product based wavelet synopses for range-sum queries. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 504–515, 2006.

[64] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 448–459, 1998.

[65] Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 101–110, 2000.

[66] N. Meratnia and R. A. de By. Spatiotemporal compression techniques for moving point objects. In *International Conference on Extending Database Technology (EDBT)*, pages 765–782, 2004.

[67] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 623–634, 2004.

[68] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 13(1):124–141, 2001.

[69] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 634–645, 2005.

[70] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(11):1451–1464, 2005.

[71] K. Mouratidis, D. Papadias, and S. Papadimitriou. Tree-based partition querying: a methodology for computing medoids in large spatial datasets. *The International Journal on Very Large Data Bases (VLDBJ)*, 17(4):923–945, 2008.

[72] S. Muthukrishnan. Data streams: algorithms and applications. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 413, 2003.

[73] S. Muthukrishnan. Subquadratic algorithms for workload-aware haar wavelet synopses. In *Proceedings of the IARCS Conference on Foundations of Software Technology and Theoretical Computer Science*, 2005.

[74] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 144–155, 1994.

[75] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 563–574, 2003.

[76] S. Papadimitriou, A. Brockwell, and C. Faloutsos. Awsom: Adaptive, hands-off stream mining. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 560–571, 2003.

[77] S. Papadopoulos, D. Sacharidis, and K. Mouratidis. Continuous medoid queries over moving objects. In *Proceedings of the International Symposium on Spatial and Temporal Databases (SSTD)*, pages 38–56, 2007.

[78] M. Potamias, K. Patroumpas, and T. Sellis. Sampling trajectory streams with spatiotemporal criteria. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 275–284, 2006.

[79] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[80] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[81] D. Sacharidis, A. Deligiannakis, and T. Sellis. Hierarchically compressed wavelet synopses. *The International Journal on Very Large Data Bases (VLDBJ)*, 2008.

[82] D. Sacharidis, K. Patroumpas, M. Terrovitis, V. Kantere, M. Potamias, K. Mouratidis, and T. Sellis. On-line discovery of hot motion paths. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 392–403, 2008.

[83] R. R. Schmidt and C. Shahabi. Propolyne: A fast wavelet-based algorithm for progressive evaluation of polynomial range-sum queries. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 664–681, 2002.

[84] C. Shahabi and R. R. Schmidt. Wavelet disk placement for efficient querying of large multidimensional datasets. Technical report, University Of Southern California, 2004.

[85] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for computer graphics: theory and applications.* Morgan Kaufmann Publishers Inc., 1996.

[86] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multidimensional uncertain data with arbitrary probability density functions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 922–933, 2005.

[87] N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic multidimensional histograms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 428–439, 2002.

[88] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 193–204. ACM Press, 1999.

[89] J. S. Vitter, M. Wang, and B. R. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 96–104, 1998.

[90] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 673–684, 2002.

[91] M. Widmann and C.Bretherton. 50 km resolution daily precipitation for the pacific northwest, 1949–94.

[92] W. Wu, W. Guo, and K.-L. Tan. Distributed processing of moving k-nearest-neighbor query on moving objects. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1116–1125, 2007.

[93] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 643–654, 2005.

[94] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 201–208, 1998.

[95] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 631–642, 2005.

[96] D. Zhang, Y. Du, T. Xia, and Y. Tao. Progressive computation of the min-dist optimal-location query. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 643–654, 2006.

[97] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 103–114, 1996.