TRCS00-09

Using Wavelet Decomposition to Support Progressive and Approximate Range-Sum Queries over Data Cubes

Yi-Leh Wu Divyakant Agrawal Amr El Abbadi

Department of Computer Science
University of California, Santa Barbara, CA 93106
Email:{ywu, agrawal, amr}@cs.ucsb.edu

Abstract

Data cubes are designed to provide aggregate information that can be used to analyze the contents of databases and data warehouses. A range query applies an aggregation operation (e.g., SUM, AVERAGE) over all selected cells in a data cube, where the selection is specified by providing ranges of values on different dimensions. Previous approaches to process the range sum query on data cubes (e.g., prefix sum) have low query costs but have high update costs. We use the discrete wavelet transformation to decompose a data cube into wavelet coefficients of different resolutions. The resulting decomposed data cube can be used for progressive and approximate query processing. This new approach can handle clustered and sparse data gracefully and can provide efficient performance for both queries and updates, which makes the wavelet decomposition technique suitable for dynamic data cube environments.

Key words. wavelet transform, data cube, query approximation, progressive query

1 Introduction

Data cubes are designed to provide aggregate information that can be used to analyze the contents of a database. To construct a data cube, the attribute of interest in the database is chosen as the *measure attribute* while other attributes are selected as the *dimension* or *functional attributes*. A data cube can be viewed as a d-dimensional array, with each dimension indexed by one functional variable and each cell containing the value of the measure attribute for the corresponding functional variable indices. Consider a retailer's database as an example. We may have *cities*, *date-of-sale*, and *productID* as the functional attributes and *sales volume* as the measure attribute. Assuming the domain of *cities* has 100 different cities indexed from 0 to 99, the domain of *date-of-sale* ranges from January 1st to December 31st, and the domain of *productID* has 1000 different products indexed from 0 to 999. The example data cube thus has $100 \times 365 \times 1000$ cells, with each cell containing the *sales volume* for the corresponding *city*, *date-of-sale*, and *productID*. A range-sum query applies an aggregation operation (e.g., SUM, AVERAGE) to all selected cells in a data cube, where the selection is specified by ranges of values for each dimension. An example of the range-sum query is *"find the total number of sales of productID between 500 and 600 at city index between 10 and 20 during the period of March 1st to March 29th"*.

Range-sum queries are useful in finding relationships and trends among attributes in a database. Efficient range-sum query processing has increased in importance with the growing interest in On-Line Analytical Processing(OLAP). Following the introduction of the data cube model by Gray et al. [1, 2], many proposals have been made to develop efficient algorithms for processing queries over data cubes [3]. However, the relational model approach in [1] is inefficient for range-sum queries (i.e., it scans the table to compute the range query) but very efficient for updates in a dynamic environment. Zhao et al. [4] suggest using multidimensional arrays to store and process data cubes, referred to as MOLAP (Multidimensional On-Line Analytical Processing). Since the space required to store the multidimensional arrays is exponential with respect to the number of dimensions in the data cube, Zhao et al. suggest to use a compressed form of *chunked* data cubes to overcome this problem. To answer a range-sum query using the multidimensional array view of data cubes, a naive way is to scan and aggregate all cells in the query ranges. The query cost of the naive scanning method is equal to the size of query ranges, which is exponential with respect to the dimensionality of the data cubes. To overcome the high range-sum query cost, Ho et al. [5] presented a simple approach to compute the range sum query efficiently. They propose to transform the original data cube into a *prefix sum* cube and process the queries on the transformed prefix sum cube. Although the prefix sum method reduces the query cost, the update cost is exponential with respect to the number of dimensions. This is because the prefix sum method maintains cumulative information in each cell. Thus if the first cell in the cube is updated, every cell in the cube will have to be updated accordingly. Recently, Geffner et al. [6, 7] have proposed modifications to the prefix-sum method to allow efficient updates to the data cube. With balanced query and update costs, this approach is therefore more suitable in a dynamic environment where the content of the database changes over time. It is also appropriate for speculative analytical processing that uses *what-if* queries.

In [14], Smith et al. propose dynamic views of data cubes using a partial sum operator pair. They state that the partial sum operator pair is equivalent to the Haar wavelet approach. By using the operator pair, users can create dynamic aggregated views. If the query patterns of users are known, the most efficient views can be generated for answering these queries with minimum extra space to store the aggregated views. This technique, however, only computes the exact query answers and does not provide estimations.

In real world applications, data cubes have usually very high dimensionality and the associated data volume is often extremely large. Due to the high dimensionality and large data volume, it becomes impractical to evaluate the exact range query results on-line. Many other algorithms have been proposed for query estimation with different precision, e.g., histogram [8], random sampling [9], Quasi-Cubes [10] wavelet transform [11], etc. Vitter et al. [11] are the first to propose using wavelet transforms on data cubes to answer range sum queries. In [11, 12, 13], wavelet transforms are used to provide quick estimations of range-sum queries. By storing only a few wavelet transform coefficients to approximate the entire data cubes, Vitter et al. achieve low space requirement and short query response time. However, the estimation techniques proposed in [11, 13] have some drawbacks. First, the approximated data cube coefficients (e.g., the wavelet coefficients used in [11]) need extra space. Second, the approximated data cube only provides estimates of given queries with designated accuracy; i.e., the accuracy of the estimates can not exceed that all the stored wavelet coefficients combined. If users need an exact answer they have to process the queries on the original data cube. Third, in a dynamic environment, it is unclear how to update the approximated data cube coefficients efficiently.

In this paper, we present a new technique for evaluating range queries in data cubes. Instead of processing queries on the original data cubes or the varieties of prefix sum data cubes, we employ wavelet decomposition of the original data cube, and perform the queries on the wavelet transform coefficient data cubes. The wavelet decomposition technique is inherently multiresolutional; i.e., it can answer range-sum queries progressively and provides multiple estimations at different resolution levels. Furthermore, the proposed technique has a desirable property that the approximate answer can be used as an input to further processing of a query for a more precise answer.

The rest of the paper is organized as follows. In Section 2, we introduce the wavelet decomposition technique on data cubes and discuss the algorithms for query and update processing using the wavelet decomposition cubes. In Section 3, we discuss the space and time complexities for different techniques on data cubes. We conclude in Section 4.

2 Wavelet-based Decomposition of Data Cubes

In this section, we start by describing the formal model of multidimensional data cubes and then extend this model to store data cubes using *wavelets*. Detailed information regarding wavelet transformations can be found in [15, 16].

2.1 Data Cube Construction using Wavelet Transformation

A data cube is basically a multidimensional array where each element of the array stores values of a measure attribute as a function of the dimensional attributes. For example, sales data may be organized in the form of a cube where the measure information is the sales volume classified in the form of a three dimensional cube with the dimensions corresponding to cities, date-of-sale, and products. More formally, let $d = \{D_1, D_2, ..., D_d\}$ denote the set of dimensions, where each dimension corresponds to a functional attribute. The d-dimensional data cube A is a d-dimensional array of size $|D_1| \times |D_2| \times ... \times |D_d|$, where $|D_i|$ is the size of dimension D_i. Without loss of generality, we assume that each array starts with index 0 and we refer to each array element as a *cell*. The total number of cells of data cube A is $N = \prod_{1 \leq i \leq d} |D_i|$.

Data cubes have the advantage that point queries correspond to indexing the appropriate cell of the multidimensional array. Furthermore, row-wise projection or column-wise projection allows users to compute fast summaries from the data cube. However, one of the most desirable properties of data cubes is that users can compute summary information from the data cube over arbitrary ranges in each dimension. A range-sum query in a d-dimensional data cube is defined as follows:

$$Sum(l_1 : h_1, ..., l_d : h_d) = \sum_{l_1 \leq i_1 \leq h_1} ... \sum_{l_d \leq i_d \leq h_d} A[i_1, ..., i_d] \tag{1}$$

In the context of data cubes, the range query specified above can be visualized as a hyperrectangle inscribed within a d dimensional cube. In raw form, computing range queries over a data cube is a function of the number of cells involved in a query hyperrectangle which in the worst case can be the entire data cube.

Alternatively, the data cube can be stored as a set of wavelet coefficients A wide variety of bases exist for wavelet decomposition, we will restrict our attention to wavelet decomposition using the Haar wavelet basis due to its simplicity, ease of implementation, and wide use. One of the primary motivations for this choice is that the Haar wavelet can be computed efficiently. To illustrate how Haar wavelets work, we start with a simple example. Suppose we have a one-dimensional function:

$$[7, 5, 9, 7] \tag{2}$$

When we perform the Haar transform on this function, we first average the function values in a pairwise fashion and thus compute a lower resolution function with pairwise average values

$$[6, 8] \tag{3}$$

and the pairwise difference values

$$[2, 2]. \tag{4}$$

The wavelet decomposition using Haar wavelet stores the pairwise differences of the original values as *detail coefficients*. It is easy to show that the original values can be recovered from the average and difference coefficients. If this process is repeated recursively on the average coefficients, then the full decomposition is as shown in Figure 1.

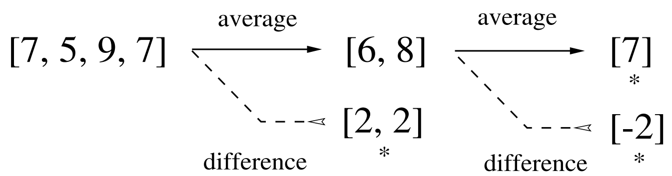


Figure 1: Wavelet Decomposition

The *wavelet transform* of the original function is one single coefficient representing the overall average followed by the detail coefficients in each increasing resolution levels. The resulting wavelet transform has thus four *wavelet coefficients*:

$$[7, -2, 2, 2] \tag{5}$$

The wavelet decomposition using Haar wavelet can be computed very efficiently, requiring only $O(N)$ time for a function with N values [11].

Note that an important property of wavelet decomposition is that no information is lost as a result of the decomposition. We can always reconstruct the original function by recursively adding and subtracting the detail coefficients from successive resolution levels. This property leads to the observation that we can use the wavelet decomposition to progressively reconstruct the data cube at multiple resolution levels while processing range sum queries. If a user is satisfied with approximate answers, then low or coarse resolution data cubes are sufficient to answer user queries. In addition, we can also provide fast response times by providing approximate results while evaluating queries at increasing resolution levels until we reach the highest resolution level.

In real world applications, data cubes have usually very high dimensionality and the associated data volume is often extremely large. Due to the high dimensionality and large data volume, it becomes impractical to evaluate the exact range query results on-line. On the other hand, it is also undesirable to provide a single approximate result with high error before the users decide to perform the exact evaluations of queries. In general, we would like to develop a system that provides continuous feedback to the users as the queries at increasing resolution levels are processed.

Another advantage of using wavelet decomposition of data cubes is that a large number of the wavelet transform coefficients are very small values, and thus they can be ignored without losing much information of the original cubes. For example sales data may have very small variations from day-to-day values within a month. In such cases, the different values are likely to be very small. Proposals have been made to ignore small values and compress the space needed to store the data cubes [12]. In this paper, however, we do not consider the general problem of data compression using wavelets.

2.2 Progressive Query Processing and Updates

The wavelet decomposition results in a hierarchical structure of wavelet coefficients at increasing resolution levels. Given a range-sum query, we can execute the query progressively starting with a low-resolution data cube and subsequently constructing the parts of the cube at a finer resolution that are necessary to answer a given query. In the following subsections, we discuss how to process range sum queries in a progressive manner, and how to update the wavelet decomposition cubes.

Figure 2 shows a two-dimensional data cube and aggregated data-cubes at lower levels of resolution. Since we are only concerned with the aggregate values for answering range sum queries, for simplicity, the difference coefficients associated with lower resolution data cubes in Figure 2 are not shown explicitly. Instead, we only depict the average value of the lower-resolution data cube for clarity. As described earlier, the average values can always be derived at each resolution in constant time.

In Figure 3, we show an example of a range sum query with range $[2:3, 0:2]$. To answer this query,

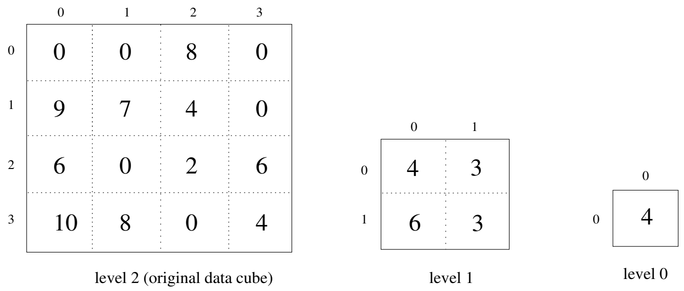


Figure 2: Sample data cube and DWT coefficients at different level

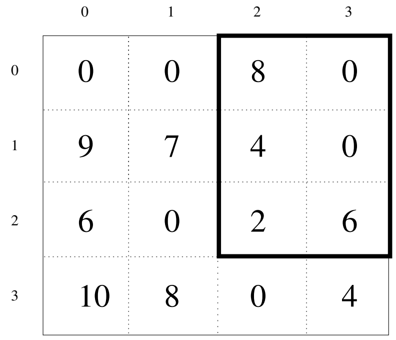


Figure 3: Query rectangle

we go through the wavelet coefficient hierarchy starting from the lowest resolution level, which is a single value that represents the average cell value of the entire cube, to the highest resolution level, where each cell contains its original value. Note that each cell represents the average of a certain portion of the data cube. For example, in Figure 2 the value of cell $[0, 0]$ at level one is four, representing the average of subcube of range $[0 : 1, 0 : 1]$. As a consequence, the total sum of the subcube of range $[0 : 1, 0 : 1]$ is $4 \times 4 = 16$. This observation implies that if the query range is $[0 : 1, 0 : 1]$, then the query can be answered by cell $[0, 0]$ at level one without going down to higher resolution level and aggregating the values of individual cells.

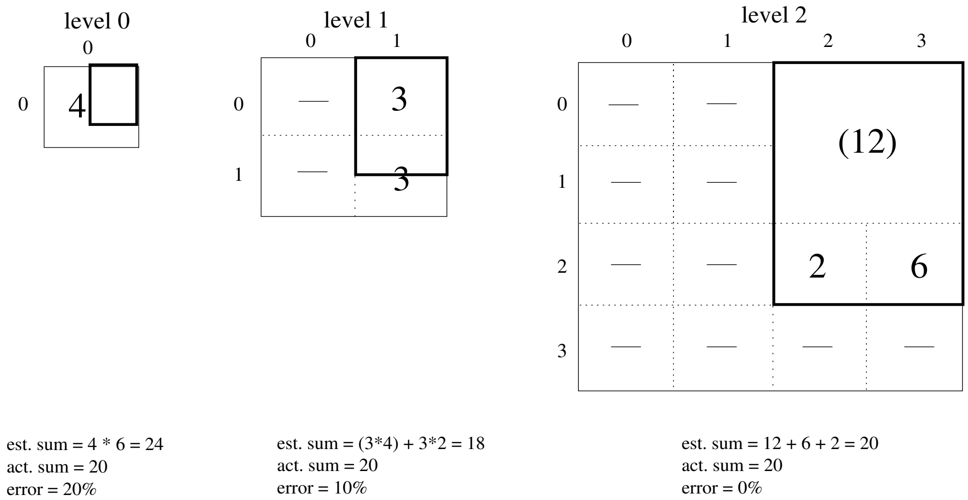


Figure 4: Progressive query at each level

Clearly, answering queries at a lower resolution and avoiding going down to the higher levels is more efficient. However, the query ranges may not be identical to a cell range at any level, as shown in Figure 3. The query range is $[2 : 3, 0 : 2]$ which is not aligned with the cube ranges at levels zero and one. We can use a *divide and conquer* strategy to answer arbitrary range-sum query using wavelet decomposition. If the query range $q = [l_1 : h_1, l_2 : h_2, ..., l_d : h_d]$ is aligned to a cell range at level n with average value c, then the answer is simply $sum(q) = c \times ((h_1 - l_1) \times (h_2 - l_2) \times ... \times (h_d - l_d))$. On the other hand, if the range q is not aligned with any cell at level n, we divide the query q into two sets. The first set contains the cells with ranges that are *fully* contained in q. The second set contains the cells with ranges that are not completely aligned with q but partially overlapped with q. For example, cell $[0,1]$ at level one has the range $[2:3, 0:1]$, which is fully contained in query range $[2:3, 0:2]$ and thus should be put in the first set. But cell $[0,2]$ at level one corresponds to the range $[2:3, 2:3]$, which only partially overlaps with query range $[2:3, 0:2]$ and hence sub-query range $[2:3, 2:2]$ should be put in the second set. The sum of range q can be answered by the sum of all subranges in sets one and two. The sum of subranges in set one can be answered at the current level n. However, the sum of subranges in set two cannot be answered at the current level n because they are partially intersected with individual cells. To answer the sum of each subrange in the second set, we create a new subquery with the specified subrange. This subquery is then computed at the higher resolution level $n+1$. We recursively apply the above steps until the query range q is fully computed.

We can give *estimated* query sums at each resolution level, assuming that the data is uniformly distributed. If the query range q is fully contained in a cell range c_q and the size of q, with the uniform distribution assumption, we can estimate the sum of range q to be $sum_{est}(q) = average(c_q) \times |q|$. Other data distribution models (i.e., normal distribution, Zipf distribution, etc.), can also be used to estimate the range sum.

Figure 4 shows the steps for answering the range sum query of Figure 3. We start at the lowest resolution level, 0, where one cell covers the entire range of the original data cube $[0:3, 0:3]$. Since the query range $[2:3, 0:2]$ is fully contained in the cell range $[0:3, 0:3]$, we can give an estimation of $sum_{est}(q) = 4 \times 6 = 24$. The true sum of range q is $8+0+4+0+2+6 = 20$, so the estimation yields $|(24-20)|/20 = 20\%$ error. Since the query range cannot be answered completely at level 0, we then try to answer it at the next higher resolution level, 1. At level 1, cell $[0,1]$ has range $[2:3, 0:1]$ and is fully contained in range q, so the actual sum of subrange $[2:3, 0:1]$ can be computed as $3 \times (2 \times 2) = 12$. However, the subrange $q_{sub1} = [2:3, 2:2]$ is not fully contained in any cell in level 1. Hence, we pass the subrange query to the next higher resolution level, 2. In the meantime, at level 1 we can give an estimation of $sum_{est}(q) = sum(cell_{(0,1)}) + sum_{est}(q_{sub1}) = 3 + 3 \times 2 = 18$. This estimate, at level one, yields a $|(18-20)|/20 = 10\%$ error, which is better than the previous estimation at level zero with 20% error. We then continue to level two and compute the sum of subrange query $q_{sub1} = [2:3, 2:2]$. Since level two is the highest resolution level, every query can be computed at this level. Two cells [2,2] and [3,2] fall completely in the query range so the $sum(q_{sub1}) = 2+6$. With the subrange $[2:4, 0:2]$ sum in level 1, we conclude that the total $sum(q) = (12) + 2 + 6 = 20$.

We note the following two properties. First, by computing partial range sums at lower resolution levels, we can reduce the number of cells needed to be aggregated to answer the complete range sum. Second, by progressively computing the range sum from lower to higher resolution levels, we can give estimates of the range sum at each level with the estimated errors reduced as the resolution level is increased. Figure 5 shows the algorithm for progressive range sum queries. It shows how to compute the exact range sum query from the wavelet coefficients cube in a progressive manner, and in the meantime gives estimates of the query result on each successive resolution level. The $BlockSum()$ function returns the range sum of one full block at current resolution level, which can be computed by the wavelet coefficients at this resolution level in constant time. The $EstimateBlockSum()$ function returns the estimation of range sum of subblock if the query range does not fully cover the whole block at this resolution level. For now we assume the simplest model that the estimate of the range sum of a subblock is simply the proportion of volume of the intersect query range and the full block; i.e., the distribution of data is assumed to be uniform.

In general, a query is divided into subqueries (i.e., finer resolution levels) only if parts of the query cannot be accurately computed at the current resolution level. If the query range is properly aligned to the boundaries of cells at the current resolution level then the answer can be computed without querying

```
PROCEDURE ProgressiveQuery (HyperCube Q)
BEGIN
    maxLevel = log N, where N is the number of cells along each dimension of the data cube
    maxEdge = Max(h_1 - l_1, h_2 - l_2, ..., h_d - l_d)
    Find level k such that maxEdge ≤ blockSize(k), i.e. maxEdge ≤ n/2^(maxLevel-k)
    Set toBeEvaluatedAtThisLevel = {Q};
    Set toBeEvaluatedAtNextLevel = NULL;
    RangeSumEstimation = RangeSumActual = 0;
    WHILE k ≤ maxLevel
        RangeSumEstimation = RangeSumActual;
        BlockSize = n/2^(maxLevel-k)
        WHILE toBeEvaluatedAtThisLevel ≠ NULL DO
            remove q from toBeEvaluatedAtThisLevel
            let q = [ql_1 : qh_1, ql_2 : qh_2, ..., ql_n : qh_n]
            boundingBlock = [⌊ql_i/BlockSize)⌋ * BlockSize : (⌊ql_i/BlockSize⌋ + 2) * BlockSize], 0 ≤ i ≤ d
            FOR EACH dimension divide into two regions:
                [⌊ql_i/BlockSize)⌋ * BlockSize : (⌊ql_i/BlockSize⌋ + 1) * BlockSize] and
                [(⌊ql_i/BlockSize⌋ + 1) * BlockSize : (⌊ql_i/BlockSize⌋ + 2) * BlockSize]
                we have 2^d such regions. DO
                let each sub-cube is sub = [sl_1 : sh_1, sl_2 : sh_2, ..., sl_d : sh_d];
                IF sub intersect with q with i
                    IF i is a full block
                        RangeSumActual+ = BlockSum();
                        RangeSumEstimation+ = BlockSum();
                    ELSE (* i is NOT a full block *)
                        add i to toBeEvaluatedAtNextLevel;
                        RangeSumEstimation+ = EstimateBlockSum();
                    ENDIF;
                ENDIF;
            END; (* FOR EACH *)
        END; (* WHILE *)
        report RangeSumEstimation at level k;
        toBeEvaluatedAtThisLevel = toBeEvaluatedAtNextLevel;
        toBeEvaluatedAtNextLevel = NULL;
        k + +;
    END; (* WHILE *)
END; (* ProgressiveQuery *)
```

Figure 5: The Algorithm for Queries

any of the finer resolution levels; the expected query cost is then in general much smaller than $O(n^d)$. But, unfortunately, in the worst case, the query cost is $O(n^d)$. The worst case happens when the query range is a very *thin* hyperrectangle of thickness one; i.e., one or more dimensions in the query range have range $[..., l_i : h_i, ...]$, where $h_i = l_i + 1$. In such cases, because the range in dimension i is one, we can only compute the exact values of each cells at the finest resolution level, which results in a worst case scenario with query cost $O(n^d)$. Similar scenarios can happen when the query range in dimension i is an odd number; i.e., $(h_i - l_i)$ is odd. Because in such cases, there is always a thin hypercube with thickness one in dimension i that cannot be aligned to cells in any of the lower resolution levels except the finest resolution level. To get the exact aggregate result, we need to process the query at the finest resolution level, which is the original data cube, and hence the query cost is $O(n^d)$. (See Appendix for a complete proof of this result.)

In real world applications, it is convenient for the users to have approximate query results fast, and then the users can decide to compute the exact query results, which takes much longer time to compute, or change the query criteria according to the approximate results. With the wavelet decomposition cubes, we can compute the queries in the few lowest resolution levels and get the approximate query results fast. Or we can modify the user queries such that the modified query ranges are aligned with the cell boundaries at lower resolution levels, which can be computed efficiently without processing through the finer resolution levels. The results are approximations of the original user queries. Note that in either cases, the approximation results can be reused for computing the actual results. Nevertheless, this algorithm can continuously provide query result estimations as it processes the queries along finer and finer resolution levels.

```
PROCEDURE Update (Point Q, Val NewValue)
BEGIN
    maxLevel = log N, where N is the number of cells along each dimension of the data cube
    OldValue = Value(P) (* get the original value at P from the data cube *)
    DiffValue = OldValue - NewValue; (* the value difference *)
    k = maxLevel − 1
    WHILE k ≥ 0
        FOR EACH coefficient block B_i, i = 1to2^d − 1
            update B_i[⌊p_1/2^(maxLevel−k)⌋, ⌊p_2/2^(maxLevel−k)⌋, ..., ⌊p_d/2^(maxLevel−k)⌋] + − = (DiffValue/k^2);
        END; (* WHILE *)
        k − −;
    END; (* WHILE *)
END; (* Update *)
```

Figure 6: The Algorithm for Updates

Figure 6 shows the detailed algorithm for update processing on data cubes. As shown in Figure 7, when updating a single cell in the original data cube, all wavelet blocks have to be updated because wavelet coefficients are summary information of different resolutions. But unlike the prefix sum method, only one cell in each wavelet coefficient block has to be updated. We can easily see that at each resolution level, at most $O(2^d)$ wavelet coefficients are dependent on the updated cell. Therefore, the overall update complexity is $O(2^d \log n)$, where n is the largest dimension size D_i. (See proof in Appendix.)

We can also show that the amount of space used by the wavelet cube is $O(N_z 2^d \log n)$ and never greater than $O(d^n)$, where n is the largest dimension size D_i and N_z is the number of nonzero data points in the original data cubes. (See proof in Appendix.) Note that the space complexity is a function of the non-zero elements N_z and not the data space. This is particularly important since in many application N_z is much less than the data space. Finally, in most applications the size of the data will be much larger than that of the available main memory, and the query processing cost will be dominated by the I/O cost rather than the CPU cost. We have developed a novel technique that can effectively reduce the I/O cost in our wavelet

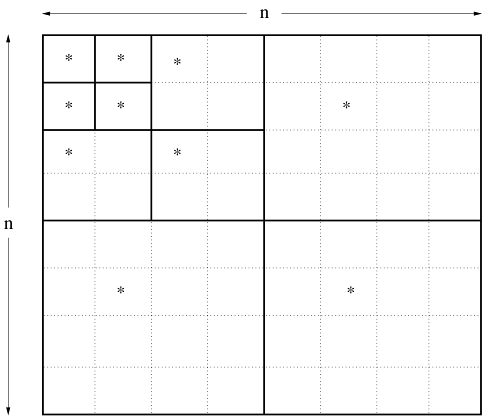


Figure 7: DWT coefficients need to be updated from a single cell update

decomposition method. Details of the storage and retrieval techniques are put in the following section.

2.3 Storage & Retrieval of Wavelet Decomposed Data Cubes

One of the most important implementation issues in real world data cube applications is large data size. In fact, we argue that the *space* is the most important issue to consider if we want to handle range sum query problem efficiently. This is because in most applications the size of the data is much larger than that of the available main memory, and the query processing cost is dominated by the I/O cost rather than the CPU cost. We start by considering the problem of wavelet decomposition of a disk-resident data cube and then address the issue of storage allocation of wavelet coefficients on secondary storage for efficient processing of progressive range-sum queries.

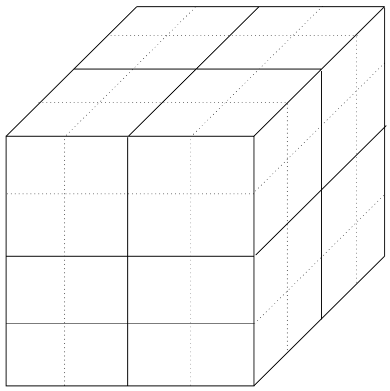
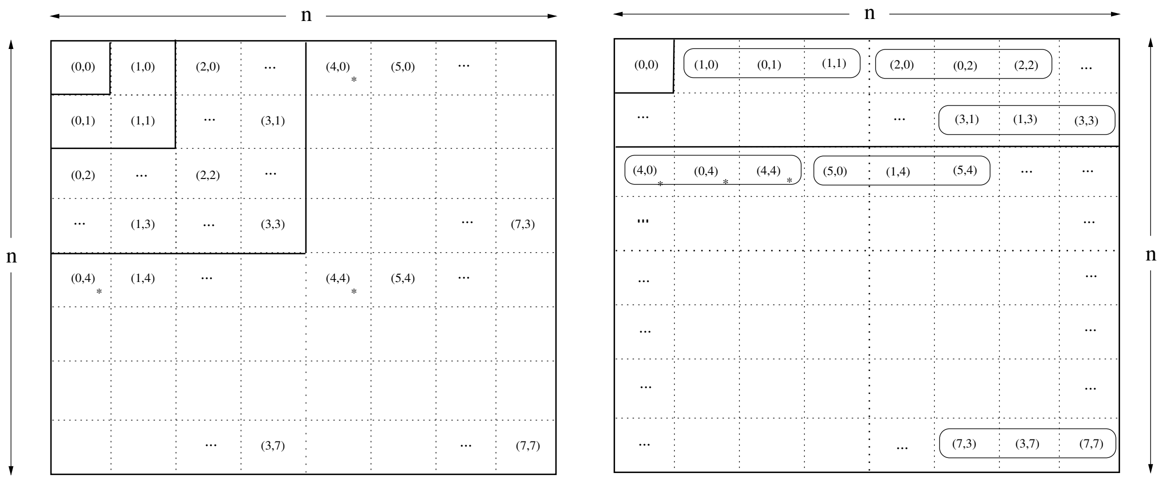


Figure 8: A chunked three-dimensional array with each chunk $2 \times 2 \times 2$

Since the complete data cube cannot be resident in the memory, we need to adapt the Haar transformation of a data cube to deal with disk resident data. In order to achieve high I/O efficiency in the presence of virtual memory, we employ the notion of *chunking* suggested by Sarawagi and Stonebraker [17] and first implemented by Zhao et al. [4] for multidimensional arrays. Figure 8 shows an example of chunking a three-dimensional array. Chunking is a way to divide a d-dimensional array into d-dimensional chunks in which each chunk is stored in one disk block. The size of each chunk is $c_1 \times ... \times c_d = B$, where c_i is the chunk size along dimension D_i and B is the disk block size. The I/O cost per chunk is just one disk I/O because we assume that the virtual memory system will handle the paging at the system level. The mapping between the indices of a cell in the original d-dimensional array and the indices of the logical combination of $\langle chunkNumber, offset \rangle$ of the cell in chunk is easy to compute since the size of dimensions and the size of a chunk along each dimension are known. Haar transforms can be easily extended to operate with chunked data cube as shown in [11].



(a) wavelet coefficients in 2d

(b) wavelet coefficients stored in row major order

Figure 9: Storage of Wavelet Coefficient Cubes

Next we focus on how to effectively reduce the I/O cost in our wavelet decomposition technique. According to the query algorithm in Section 2.2, at each resolution level, exactly $O(2^d)$ wavelet coefficients have to be retrieved to compute the block sum at this level, which implies that we need $O(2^d)$ I/O to retrieve all coefficients if they are located in different disk blocks, which is usually the case as in Figure 9(a). Note that all of the previous works on prefix sum cubes suffer from the same problem. However, since the wavelet decomposition technique always decomposes the cube data in 2^d chunk at *any* resolution level, we can simply reposition the wavelet coefficients, such that all the 2^d coefficients needed to evaluate a block sum at every level reside in the same disk block, as shown in Figure 9(b). In this case, only one disk I/O is needed to retrieve all wavelet coefficients needed to compute the block sum at each level. This simple technique yields a $O(2^d)$ speed up not only for the range-sum query processing but also for the update processing. More significantly, by this simple technique, all updates have a fix upper bound of $O(\log N)$ disk I/Os, which makes the wavelet decomposition technique more attractive for dynamic data cubes with frequent updates.

The space problem becomes more acute as the dimensionality of data cube grows. For example, if the dimensionality is high (e.g., 20 dimensions) which is reasonable in real world applications, and if the cardinality of each dimension is ten, i.e., ten cells along each dimension, then the total number of cells in the data cube is 10^{20}. This value is too large to fit in any computer system memory today. This observation implies that if the dimensionality of data cube is high, the data cube must be sparse. This sparseness property makes the prefix sum technique [5] inappropriate for the range sum query processing in the high dimensional data cube. This is because the prefix sum method maintains cumulative information in each cell. Thus if the first cell in the cube is non-zero, every cell in the cube will have a non-zero value.

Two properties of the wavelet transform make it suitable for the range-sum query problem. First, the compression property makes it possible to store the data cube in a much *smaller* space without losing any information of the original data cubes. All previous methods have space requirements equal to the original data cubes or larger if the additional summary information is stored. But as we stated above, even the hope of using the same space of the original data cubes is not feasible. Some form of compression is necessary to reduce the space requirements. Our wavelet decomposition technique is suitable because of the implied compression capability. Second, the multiresolutional property of wavelet decomposition makes the progressive query evaluation possible, which is a vital capability if the user queries cannot be evaluated in a short time. Furthermore, the processing cost is exponential (either 2^d or n^d) with respect to the dimensionality in all proposed techniques. Thus, in most cases, users cannot expect interactive response times. Some form of approximations or estimations becomes necessary to circumvent these problems. We rely on the multiresolution nature of wavelets to provide approximate and progressive answers to user queries.

3 Comparison of Different Cube Implementations

In this section, we compare the proposed wavelet approach with other data cube implementations. We start by analyzing the complexity of these implementations, and then conduct some experimental comparisons.

3.1 Space and Processing Complexity

Although many estimation techniques have been developed for the range sum queries in data cubes, we still often need to compute the exact answers for arbitrary range sum queries. There are two major categories of techniques for this purpose, the ROLAP (Relational On-Line Analytical Processing) [1], which stores the data in the relational model as tables, and the MOLAP (Multidimensional On-Line Analytical Processing) [4], which stores the data in multidimensional arrays.

symbol	meaning	range
d	no. of functional attributes (i.e., no. of dimensions in data cubes)	2^1 to 2^5
n	without loss of generality, no. of possible values for ALL functional variables	2^1 to 2^{10}
N_Z	no. of non-zero input data tuples (i.e., no. of non-zero points in data cubes)	10^3 to 10^9
k	no. of wavelet coefficients stored	2^4 to 2^{10}

Table 1: Symbols used for space requirement analysis

Table 1 shows the definitions of symbols used through this section for space requirement analysis.

cost	Query (exact/estimate)	Update	Storage
original data cube (relational)	$O(Nz)/--$	$O(N_z)$	$O(Nz \times (d+1))$
original data cube(array) [4]	$O(n^d)/--$	$O(1)$	$O(n^d)$
prefix sum [5]	$O(2^d)/--$	$O(n^d)$	$O(n^d)$
RPS Tree [6]	$O(\log^d n)/--$	$O(\log^d n)$	$O(n^d) + RPStree$
wavelet [13]	$--/O(2dk)$	$--$	$O(k(d+1))$
proposed technique	$O(n^d)/O(n^d)$	$O(2^d \log n)$	$O(N_z 2^d \log n)$

Table 2: Cost Comparison of Different Range Sum Query Techniques

Table 2 shows the cost comparison of different techniques. Without loss of generality, we assume that the space needed to store each attribute value is one storage unit. If we use the ROLAP model, the space required to store each input data point will be the space needed to store d functional variables in addition to the space to store one measurement variables. So the total space needed to store the input data is $O(N_z \times (d+1))$, which is the number of input tuples, N_Z, multiplied by the space needed to store each tuple. As for query and update cost, it depends whether an index is used or not, but in general they are $O(N_Z)$ (as the number of dimension increases, the benefit of index structures, in general, decreases.) The storage requirement to store the data cube in multidimensional array is $N_1 \times N_2 \times ... \times N_d = O(n^d)$ without using the compressed form of *chunked* cubes suggested by [4]. Query cost is a function of the size of the query, hence a query of size n_q in each dimension causes query cost of complexity $O({n_q}^d)$, hence in the worst case $O(n^d)$. Update costs, on the other hand, are constant $O(1)$.

The prefix sum cubes technique [5] can process arbitrary range sum queries in constant time (i.e., $O(2^d)$), regardless of the size or shape of the query ranges. Although the query processing cost is reduced to the minimum, constant time, we argue that the storage requirement for the prefix sum technique is not practical. The storage requirements to store the prefix sum cubes are the same as the original data cubes. Note, however, that in prefix sum cubes, each cell stores the sum of all cells that fall into the range from the origin cell (i.e., all the dimension indices are zero) to the current cell. In the worst case, if the origin cell is updated, then the entire prefix sum cube needs to be updated, which results in an update cost of $O(n^d)$. Due to the exponential cost associated with updates, prefix sum cubes are not suitable for dynamic environments. The RPS Tree [6] technique overcomes this problem and achieve $O(\log^d n)$ for both query and update.

The wavelet based technique proposed in [13] can compute estimates of given queries in $O(2dk)$ time. However, the exact query result can not be computed by the stored wavelet coefficients and the update cost of the stored wavelet coefficients is unclear.

3.2 Evaluation

In this section, we conduct several experiments to verify the applicability of the proposed wavelet decomposition method compared to the prefix sum method. In the first experiment we compare the query and update costs of the proposed method and the prefix sum method for different data cubes. Since the prefix sum method cannot give estimates of the query result but only the exact result, we are forced to compare the query cost of computing the exact result with both methods. We use synthesized data cubes with different dimensionality, range from four to eight, where the domain size (i.e., number of cells) along each dimension is fixed to 16. The size of the data cubes range from 2^{16} to 2^{32}. We performed 1000 independent range sum queries and updates to compute the average query and update costs for each method. The query range for each query is chosen randomly along each dimension independently. Similarly, the cell to be updated is chosen randomly along each dimension. Since both methods can be implemented in various ways (in terms of different disk allocation strategies, etc.), which effect the actual run time behavior, to get around the detailed implementation issues, we use as the cost metric the number of cells that need to be accessed or updated for each query or update. Note that with this cost metric, the query and update costs of the prefix sum method are independent of the actual data distribution in the data cubes. But for the wavelet decomposition method, the query cost depends on the density of the data cubes, i.e., the sparser the data cubes, the lower the query costs. We choose to ignore this difference and assume dense data cubes and thus give the worst case scenario query cost for the wavelet decomposition method.

Figure 10 shows the results of the first experiment. Note how the query and update costs scale as the dimension increases. The prefix sum method has the smallest query cost, which is of $O(2^d)$ in any dimension, independent of the query ranges. On the contrary, the update cost of the prefix sum method is the largest in any dimension, which is of $O(n^d)$. As a result, we can see that the prefix sum method is more suitable in near static data cube applications, many queries but very few updates. On the other hand, the proposed wavelet decomposition method has higher query costs but lower update costs than the prefix sum method.

The next experiment explores the progressive property of the proposed wavelet decomposition method. As mentioned earlier, our proposed method is the only technique that can give multiple estimates of the query results progressively as the query is being processed through multiple wavelet resolutions. We use real data in this experiment to show the applicability of the proposed method in real world applications. The data used in this experiment is the DES data taken from the US Census Bureau. Out of hundreds of attributes in the DES data, we choose six attributes , namely *age*, *education*, *marital status*, *race*, *sex*, and *household summary*, as the functional attributes and *personal income* as the measure attribute. We normalize the six functional attributes so that each attribute has the same domain size of 16. The resulting data cube is a 6-dimensional cube with each dimension of 16 cells, which is 2^{24} cells in total. The number of

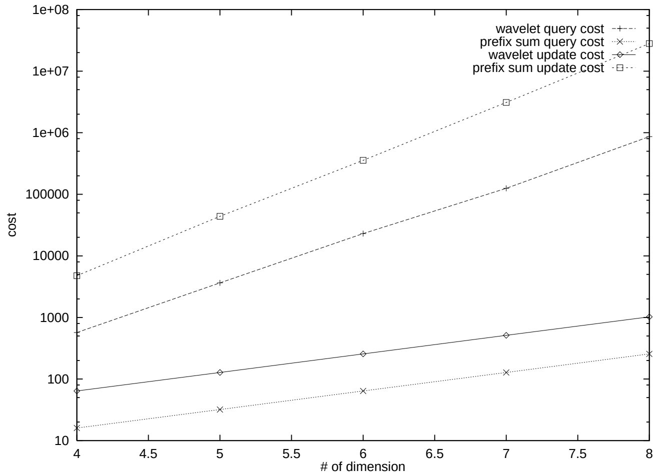


Figure 10: Query and update cost comparison

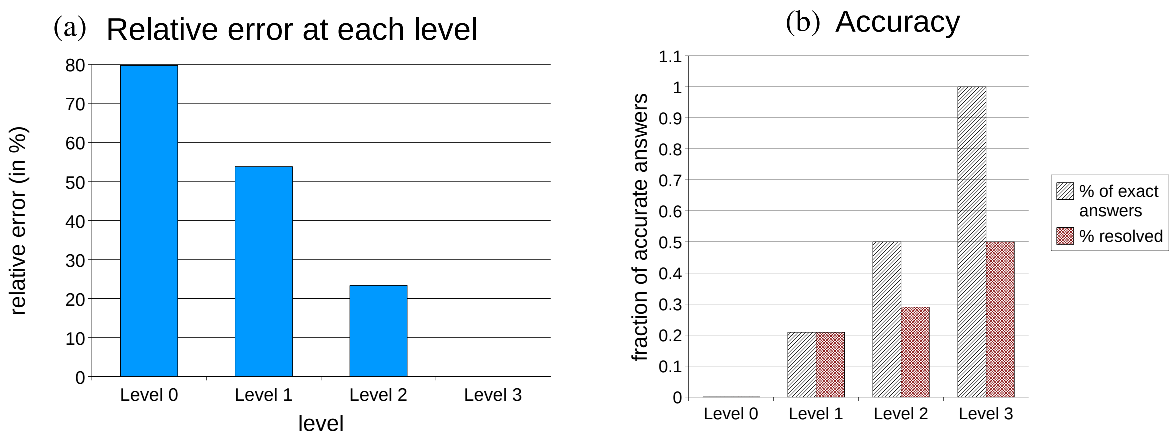


Figure 11: Wavelet decomposition method using census data

tuples in the original data set is 149642, which yields the density of the data cube slightly less then 1%. We perform 1000 random queries with the query ranges chosen randomly along each dimension independently. For each query, the estimated query result s' approximates the exact query result s, we defined the *absolute error* of the query to be the absolute difference between s and s', $|s - s'|$. The *relative error* E_{rel}, which evaluates the error as related to the exact result is defined as $E_{rel} = \frac{|s-s'|}{s} \times 100$. A problem that has been observed [12] with E_{rel} is that when the exact query result is zero, the relative error E_{rel} become infinity. For computational reasons, in such cases, we readjust E_{rel} from infinity to 100%.

Figure 11.a shows the average relative error of the 1000 queries in different resolution levels, where level 0 is the lowest resolution level. Note that the average relative error decreases as the resolution increases. Users can stop the progressive query processing at any level if they are satisfied with the estimates. Or they can wait until the query has been processed at the highest resolution level that is needed, which yields the exact query result.

In many cases, we do not need to process the queries in the highest resolution level to compute the exact query results. This is typically the case when the exact query result is zero; i.e., all cells contained in the query range have zero values. If we learn that the query result at level i is zero then we need not process the query at the finer resolution level $i + 1$. Another case is when the query range is aligned to the cell boundaries at level i, then the total aggregate value of the range can be computed by the average value at this resolution level without further processing in finer resolution levels. Figure 11.b shows the percentage of the 1000 random queries that can be fully evaluated at each resolution level. The areas darkly shadowed are the percentage of queries that can be fully evaluated at this resolution level but *NOT* in coarser resolution level. For example, Figure 11.b shows that nearly 30% of the queries can be fully evaluated at resolution level 2 but not in level 0 or 1. The areas lightly shadowed are the aggregated percentage of queries that can be fully evaluated at this resolution level *OR* in coarser resolution level. Figure 11.b shows that about 50% of the queries can be fully evaluated at resolution level 2 without processing them in the finest resolution level. This result suggests that only half of the queries need to be processed in the finest resolution level to get exact query results, which may yield the worst case scenario query cost $O({n_q}^d)$ as described in Section 3.1. The rest of the queries can be processed in coarser resolution levels, thus yielding significantly less query costs than $O(n^d)$. This multiresolutional property, which yields small average query cost, combined with the progressive processing property, make the proposed wavelet decomposition method more suitable for real world dynamic OLAP applications.

4 Concluding Remarks

In this paper, we presented a wavelet decomposition method for evaluating range-sum queries in data cubes. The advantage of the wavelet decomposition method is the multiresolution property of discrete wavelet transform. This property enables progressive and approximate evaluation of range-sum queries of the data cube. Another desirable property of the proposed approach is that approximate answer can be used as an input to further processing of a query for a more precise answer. As discussed in Section 3, query processing over high dimensional sparse data cube remains a significant problem. New approaches are needed to deal with sparse data cubes.

References

[1] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Procedings of the 12th IEEE Conference on Data Engineering - ICDE 96*, pages 152–159, 1996.

[2] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, and Murali Venkatrao. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Data Mining and Knowledge Discovery 1*, pages 29–53. Kluwer Academic Publishers, 1997.

[3] A. Agrawal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *Procedings of the 22nd Intl. Conf. Very Large Database*, pages 506–521, Mumbai(Bombay), India, September 1996.

[4] Yihong Zhao, Prasad M DeshPande, and Jeffrey F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Procedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 159–170, Arizona, 1997.

[5] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range Queries in OLAP Data Cubes. In *Procedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 73–88, Arizona, 1997.

[6] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative Prefix Sums: And Efficient Approach for Querying Dynamic OLAP Cubes. In *Procedings of the International Conference on Data Engineering - ICDE 99*, pages 328–335, Sydney, Australia, March 1999.

[7] S. Geffner and D. Agrawal and A. El Abbadi. Range Sum Queries in Dynamic Data Cubes. Technical report, Computer Science Department, University of California at Santa Barbara, Santa Barbara, California, 1999.

[8] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Procedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, May 1996.

[9] P. Haas and A. Swami. Sampling-based Selectivity for Joins Using Augmented Frequent Value Statistics. In *Procedings of the 1995 ACM SIGMOD International Conference on Management of Data*, March 1995.

[10] Daniel Barbara and Mark Sullivan. Quasi-Cubes: A Space-Efficient Way to Support Approximate Multidimensional Databases. Technical Report ISSE-TR-98-03, School of Information Technology and Engineering, George Mason University, Fairfax, Virginia, October 1998.

[11] Jeffrey Scott Vitter, Min Wang, and Bala Iyer. Data Cube Approximation and Histograms via Wavelets. In *Proceedings of Seventh International Conference on Information and Knowledge Management - CIKM'98*, pages 96–104, Bethesda, Maryland, November 1998.

[12] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-Based Histograms for Selectivity Estimation. In *Procedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 448–459, Seattle, 1998.

[13] Jeffrey Scott Vitter and Min Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In *Procedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 193–204, Phildelphia, June 1999.

[14] John R. Smith, Chung-Sheng Li, Vittorio Castelli, and Anant Jhingran. Dynamic assembly of views in data cubes. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 274–283. ACM Press, 1998.

[15] C. Sidney Burrus, Remesh A. Gopinath, and Haitao Guo. *Introduction to Wavelets and Wavelet Transforms: A Primer.* Prentice-Hall, Inc., 1998.

[16] Eric J. Stollnitz, Tony D. Derose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications.* Morgan Kaufmann Publishers, Inc., 1996.

[17] S. Sarawage and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *Procedings of the 11th IEEE Conference on Data Engineering - ICDE 94*, Houston, Taxas, 1994.

Appendix: Proof of Theorems

Theorem 4.1 *In the worst case, the wavelet decomposition cube has range sum query complexity $O(n^d)$, where n is the largest dimension size D_i.*

Proof. The range-sum query is $Sum(l_1 : h_1, l_2 : h_2, ..., l_i : h_i, ..., l_d : h_d)$ where $l_i \leq h_i, 1 \leq i \leq d$. We can always find a level k in the wavelet coefficient tree such that the query range is smaller than or equal to the block size at level k, i.e., $Max(h_1 - l_1, h_2 - l_2, ..., h_d - l_d) \leq 2^{MaxLevel-k}$, where $MaxLevel$ is the height of the multiresolutional coefficient tree and is equal to $\log N$. Figure 12 shows all the possible cases that the query rectangles intersect the blocks at level k in a two dimensional view. In any of the above cases, we can decompose the query rectangle into at most 2^d sub-hypercubes by taking the intersections of the query hypercubes to each block at level k. If one of the sub-hypercubes is exactly one block size then it can be computed in $O(1)$ time. As shown in Figure 12, the subrectangle E can be derived in constant time by taking the wavelet coefficient representing that block times the block size. Then we compute the rest of the $2^d - 1$ subrectangles by recursively querying the data cube at the next resolution level $k + 1$. Hence, the worst case cost of an arbitrary range-sum query is a series:

$$QueryCost(Q) = \overbrace{2^d[2^d[2^d....]]}^{\log n\ times} = O(2^{d^{\log n}}) = O(n^d) \tag{6}$$

□

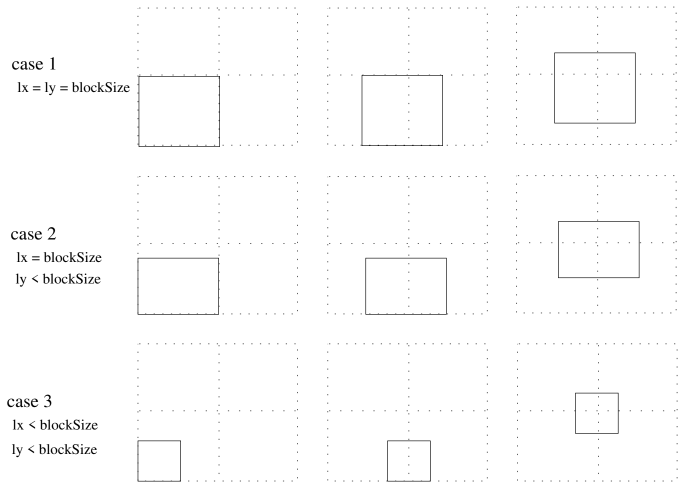


Figure 12: 2 dimensional case

Theorem 4.2 *The wavelet data cube has update complexity of $O(2^d \log n)$, where n is the largest dimension size D_i.*

Proof.

When updating a single cell in the original data cube, all wavelet blocks have to be updated because wavelet coefficients are "summary" information of different resolutions. But unlike the prefix sum method, only one cell in each wavelet coefficient block has to be updated. So the cost of update is the same as the number of wavelet coefficient blocks. For each resolution level, the number of wavelet coefficient blocks remains the same as 2^d. And one of the coefficient block, the summation block, will be decomposed again. So at each resolution level there are exactly $2^d - 1$ coefficient blocks. The total update cost is the same as the total coefficient blocks at each levels, that is,

$$UpdateCost(Q) = (2^d - 1) * \log n + 1 = O(2^d \log n) \tag{7}$$

□

Theorem 4.3 *The wavelet data cube has space complexity of* $O(N_z 2^d \log n)$, *where n is the largest dimension size* D_i *and* N_Z *is the number of nonzero data points in the original data cubes.*

Proof. As described in the proof of Theorem4.2, at each resolution level there are exactly $2^d - 1$ coefficient blocks for each nonzero data point in the original data cubes. The number of resolution levels in data cubes is $\log n$, where n is the largest dimension size D_i. The total number of coefficients in the wavelet decomposition cube is $N_Z \times (2^d - 1) \times \log n = O(N_Z 2^d \log n))$, where N_Z is the number of nonzero data points in the original data cubes. □