Accelerating Multi Dimensional Queries in Data Warehouses

Russel Pears and Bryan Houliston

School of Computing and Mathematical Sciences,
Auckland University of Technology, New Zealand
Email rpears@aut.ac.nz

Data Warehouses are widely used for supporting decision making. On Line Analytical Processing or OLAP is the main vehicle for querying data warehouses. OLAP operations commonly involve the computation of multidimensional aggregates. The major bottleneck in computing these aggregates is the large volume of data that needs to be processed which in turn leads to prohibitively expensive query execution times. On the other hand, Data Analysts are primarily concerned with discerning trends in the data and thus a system that provides approximate answers in a timely fashion would suit their requirements better.

In this chapter we present the Prime Factor scheme, a novel method for compressing data in a warehouse. Our data compression method is based on aggregating data on each dimension of the data warehouse.

Extensive experimentation on both real-world and synthetic data have shown that it outperforms the Haar Wavelet scheme with respect to both decoding time and error rate, while maintaining comparable compression ratios (Pears and Houliston, 2007). One encouraging feature is the stability of the error rate when compared to the Haar Wavelet. Although Wavelets have been shown to be effective at compressing data, the approximate answers they provide varies widely, even for identical types of queries on nearly identical values in distinct parts of the data. This problem has been attributed to the thresholding technique used to reduce the size of the encoded data and is an integral part of the Wavelet compression scheme. In contrast the Prime Factor scheme does not rely on thresholding but keeps a smaller version of every data element from the original data and is thus able to achieve a much higher degree of error stability which is important from a Data Analysts point of view.

Keywords: Data Warehousing, OLAP, Multi Dimensional Aggregates, Approximate Query Processing, Wavelets, Prime Numbers, Data Compression.

INTRODUCTION

Data Warehouses are increasingly being used by decision makers to analyze trends in data (Cunningham, Song and Chen, 2006, Elmasri and Navathe, 2003). Thus a marketing analyst is able to track variation in sales income across dimensions such as time period, location, and product on their own or in combination with each other. This analysis requires the processing of multi-dimensional aggregates and group by operations against the underlying data warehouse. Due to the large volumes of data that need to be scanned from secondary storage, such queries, referred to as On Line Analytical Processing (OLAP) queries, can take from minutes to hours in large scale data warehouses (Elmasri, 2003, Oracle 9i).

The standard technique for improving query performance is to build aggregate tables that are targeted at known queries (Triantafillakis, Kanellis, and Martakos 2004; Elmasri, 2003). For example the identification of the top ten selling products can be speeded up by building a

summary table that contains the total sales value (in dollar terms) for each of the products sorted in decreasing order of sales value. It would then be a simple matter of querying the summary table and retrieving the first ten rows. The main problem with this approach is the lack of flexibility. If the analyst now chooses to identify the bottom ten products an expensive re-sort would have to be performed to answer this new query. Worst still, if the information is to be tracked by sales location then the summary table would be of no value at all. This problem is symptomatic of a more general one where Database Systems which have been tuned for a particular access pattern perform poorly as changes to such patterns occur over a period of time. In their study (Zhen and Darmont, 2005) showed that database systems which have been optimized through clustering to suit particular query patterns rapidly degrade in performance when such query patterns change in nature.

The limitations in the above approach can be addressed by a data compression scheme that preserves the original structure of the data. The chapter is organized as follows. In the next section we review related work. The next section introduces the Prime Factor Compression (PFC) approach. We then present the algorithms required for encoding and decoding with the PFC approach. The On Line reconstruction of Queries is discussed thereafter. Implementation related issues are then discussed, followed by a performance evaluation of PFC and a comparison with the Haar Wavelet algorithm. We then discuss future trends in optimizing multi-dimensional queries in the light of the results of this research. We conclude with a summary of the main achievements of the research.

BACKGROUND

Previous research has tended to concentrate on computing exact answers to OLAP queries (Ho, and Agrawal, 1997, Wang 2002). Ho describes a method that pre-processes a data cube to give a prefix sum cube. The prefix sum cube is computed by applying the transformation: $P[A_i]=C[A_i]+P[A_{i-1}]$ along each dimension of the data cube, where P denotes the prefix sum cube, C the original data cube, A_i denotes an element in the cube, and i is an index in a range $1..D_i$ (D_i is the size of the dimension D_i). This means that the prefix cube requires the same storage space as the original data cube.

The above approach is efficient for low dimensional data cubes. For high dimensional environments, two major problems exist. Firstly, the number of accesses required is 2^d (Ho *et al*, 1997), which can be prohibitive for large values of d (where d denotes the number of dimensions). Secondly, the storage required to store the prefix sum cube can be excessive. In a typical OLAP environment the data tends to be massive and yet sparse at the same time. The degree of sparsity increases with the number of dimensions (OLAP) and thus the number of non zero cells may be a very small fraction of the prefix sum cube, which by its nature has to be dense for its query processing algorithms to work correctly.

Another exact technique is the Dwarf cube method (Sismannis and Deligiannakis, 2002) which seeks to eliminate structural redundancies and factor them out by coalescing their store. Pre-fix redundancy arises when the fact table contains a group of tuples having a prefix of redundant values for its dimension attributes. On the other hand, suffix redundancy occurs when groups of tuples contain a suffix of redundant values for its dimension attributes. Elimination of these redundancies has been shown to be effective for dense cubes. Unfortunately, in the case of sparse cubes with a large number of dimensions the size of the fact table can actually increase in size (Sismannis *et al*, 2002), thus providing no gains in storage efficiency.

In contrast to exact methods, approximate methods attempt to strike a good balance between the degree of compression and accuracy. Query performance is enhanced by storing a compact version of the data cube. A histogram based approach was used by (Matias and Vitter, 1998), (Poosala and Gnati, 1999), (Vitter *et al*, 1998), to summarize the data cube. However, histograms too suffer from the curse of high dimensionality, with both space and time complexity increasing exponentially with the number of dimensions (Matias *et al*, 1998).

The Progressive Approximate Aggregate approach (Lazaridis and Mehrotra, 2001) uses a tree structure to speed up the computation of aggregates. Aggregates are computed by identifying tree nodes that intersect the query region and then accumulating their values incrementally. All tree nodes that are fully contained in the query region provide an exact contribution to the query result, whereas nodes that have a partial intersection provide an estimate. This estimation is based on an assumption of spatial uniformity of attribute values across the underlying data cube. In practice this assumption may be invalid as with the case of the real-world data (US Census) that we experimented with in our study. In contrast, our method makes no such assumptions on the shape of the source data distribution. Another issue with the above scheme is that it has a worst case run time performance that is linear in the number of data elements covered by the query, as observed by (Chen and Chen, 2003). This has negative implications for queries that span a large fraction of the underlying data cube, particularly since compression is not utilized to store source data.

Sampling is another approach that has been used to speed up aggregate processing. Essentially, a small random sample of the database is selected and the aggregate is computed on this sample. The sampling operation can be done off-line, whereby the sample is extracted from the database and all queries are run on this single extracted sample. On the other hand, in on-line sampling data is read randomly from the database each time a query is executed and the aggregate computed on the dynamically generated sample (Hellerstein *et al*, 1996). The very nature of sampling makes it very efficient in terms of run time, but its accuracy has been shown to be a limiting factor in its widespread adoption (Vitter and Wang, 1999).

Vitter *et al*, use the wavelet technique to transform the data cube into a compact form. It is essentially a data compression technique that transforms the original data cube into a Wavelet Transform Array (WTA) which is a fraction of the size of the original data cube. In their research Matias *et al* show that wavelets are superior to the histogram based methods, both in terms of accuracy and storage efficiency. Wavelets have also been shown to provide good compression with sparse data cubes, unlike the Dwarf compression method.

Given the wavelet's superior performance over its rivals and the fact that it is an approximate technique, it was an ideal choice for comparison against our Prime Factor scheme which is also approximate in nature. The next section provides a brief overview of the encoding and decoding procedures used in wavelet data compression.

In principle, any data compression scheme can be applied on a data warehouse For example, a 3-dimensional warehouse that tracks sales by time period, location and products can be compressed along all three dimensions and then stored in the form of "chunks" (Sarawagi and Stonebraker, 1994). Chunking is a technique that is used to partition a d-dimensional array into smaller d-dimensional units.

However a high compression ratio is needed to offset the potentially huge secondary storage access times. This effectively ruled out standard compression techniques such as Huffman Coding (Cormack, 1985), LZW and its variants (Lempel and Ziv, 1977; Hunt 1998) as well as

Arithmetic Coding (Langdon, 1984). These schemes enable decoding to the original data with 100% accuracy, but suffer from modest compression ratios (Ng and Ravishankar, 1997). On the other hand the trends analysis nature of decision making means that query results do not need to reflect 100% accuracy. For example, during a drill-down query sequence in ad-hoc data mining, initial queries in the sequence usually determine the truly interesting queries and regions of the database. Providing approximate, yet reasonably accurate answers to these initial queries gives users the ability to focus their explorations quickly and effectively, without consuming inordinate amounts of valuable system resources (Hellerstein, Haas and Wang, 1997).

This means that lossy schemes which exhibit relatively high compression and near 100% accuracy would be the ideal solution to achieving acceptable performance for OLAP queries. This chapter investigates and presents the performance of a novel scheme, called Prime Factor Compression (PFC) and compares it against the well known Wavelet approach (Vitter and Wang, 1998; Vitter and Wang 1999, Chakrabarti and Garofalakis, 2000). Recent results have indicated that the Prime Factor Compression scheme outperforms the Wavelet scheme in terms of error stability, maintaining a very low and virtually constant level of accuracy irrespective of the size of the query (Pears and Houliston, 2007). This is in marked contrast to the Wavelet scheme which exhibits large swings in accuracy with varying query size. This problem has been attributed to the thresholding technique used to reduce the size of the encoded data (Garofalakis and Gibbons, 2004) and is an integral part of the Wavelet compression scheme. The Prime Factor scheme, on the other hand does not rely on thresholding but keeps a smaller version of every data element from the original data and is thus able to achieve a much higher degree of error stability which is important from a Data Analysts point of view.

In this chapter we provide a fuller exploration of the PFC scheme, including detailed results on various different types of datasets and a formal evaluation of its performance on sparse data.

Wavelet Data Compression Scheme

The Wavelet scheme compresses by transforming the source data into a representation (the Wavelet Transform Array or WTA) that contains a large number of zero or near-zero values. The transformation uses a series of averaging operations that operate on each pair of neighboring source data elements. In this manner the original data is transformed into a data set (the Level 1 transform set) that contains the averages of pairs of elements from the original data set. The pair-wise averaging process is then applied recursively on the Level 1 transform set. The process continues in this manner until the size of the transformed set is equal to 1. In order to be able to reconstruct the original data the pair-wise differences between neighbors is kept in addition to the average. The WTA array then consists of all pair-wise averages and differences accumulated across all levels.

A thresholding function is then applied on the WTA to remove a large fraction of array elements which are small in value. The thresholding function applies a weighting scheme on the WTA elements as those elements at the higher levels play a more significant role in reconstruction than their counterparts at the lower levels. For details of the wavelet encoding and decoding techniques the reader is referred to (Vitter *et al*, 1999).

Wavelet Decoding

The decoding process reconstructs the original data by using the truncated version of the WTA. The decoding process is best illustrated with the following example. Figure 1 shows how the

coefficients of the original array are reconstructed from the WTA (the C coefficients hold the original array while the S coefficients hold the WTA).

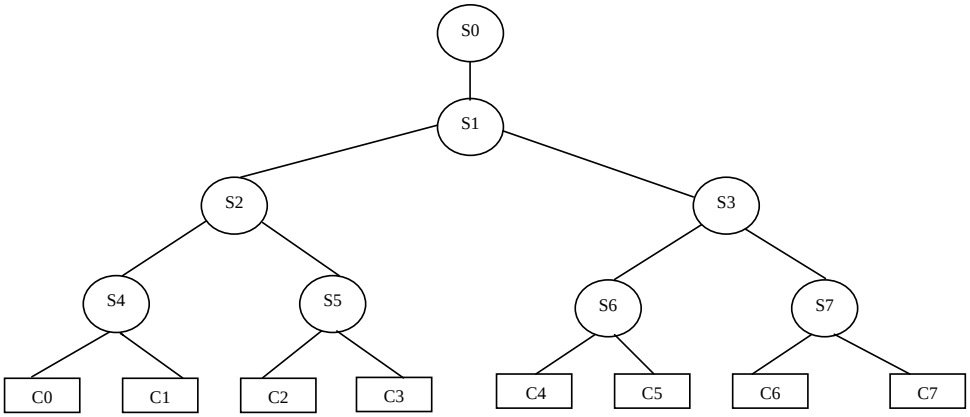


Figure 1: The Wavelet Reconstruction Process

Any coefficient C(i) is reconstructed by using its ancestor S coefficients in the path from the root node to itself. Thus for example, C(0) = S(0) +S(1)+S(2) +S(4) and C(1) = S(0)+S(1)+S(2)-S(4).

Consider a scenario where S(4), although relatively large in comparison to S(0), S(1) and S(2), is thresholded out due to its lower weighting, thus leading to inaccuracies in the estimation of C(0) and C(1). This is symptomatic of the general case where the lower level coefficients are significant contributors to the reconstruction process but are thresholded out to make way for their more highly weighted ancestor coefficients. The problem is especially acute in the case of data that is both skewed and have a high degree of variability.

THE PRIME FACTOR SCHEME

In response to the problems associated with wavelets, we present the “Prime factor Scheme”. The scheme works broadly on the same lines as the Wavelet technique in the sense that data compression is used to reduce the size of the data cube prior to processing of OLAP queries. OLAP queries are run on the compressed data, which is decoded to yield an approximate answer to the query.

Our encoding scheme uses pre-processing to reduce the degree of variation in the source data which results in better compression. An overview of the encoding process is given in the following section.

Overview of Prime Factor Encoding Scheme

The data is first scaled using the standard min-max scaling technique . With this technique, any value V in the original cube is transformed into V’, where V’ =(V-min)*(nmax-nmin)/(max-min)+nmin, where min, max represent the minimum and maximum values respectively in the original data cube; nmin and nmax are the corresponding minimum and maximum values of the

scaled set of values. Each scaled value V' is then approximated by its nearest prime number in the scale range [nmin, nmax].

The choice of nmin and nmax influences both the degree of compression and the error rate as we shall show later in the "Experimental Setup" section. The rationale behind the scaling is to induce a higher degree of homogeneity between values by compressing the original scale [min, max] into a smaller scale [nmin, nmax], with nmin ≥ min and nmax < max. In doing so, this pre-processing step improves the degree of compression.

The scaled data cube is then partitioned into equal sized chunks. The size of the chunk c represents the number of cells that are enclosed by the chunk. The size of the chunk affects the decoding (query processing) time. Higher values mean fewer chunks need to be decoded, thus reducing the decoding time (see Theorem 3 in the section on "On line Reconstruction of Queries").

Each chunk is encoded by the prime factor encoding algorithm which yields an array containing 2c cells. Although the encoded version has twice the number of cells it is much smaller in size since each cell is very small in numerical value. In fact, our experimental results reveal that the vast majority of values are very small integers (see Figures 13a and 13b in the "Experimental Setup" section). Figure 2 below summarizes the Prime Factor encoding process.

In addition to transforming values to very small integers, the other major benefit of the algorithm is that the integers are highly skewed in value towards zero. For example, on the Census data set (US Census) with a chunk size of 64, over 75% of the values turned out to be zero. These results were also borne out with the synthetic data sets that we tested our scheme on. The source (original) data in some of these data sets were not skewed in nature, showing that the skew was induced, rather than being an inherent feature of the original data.

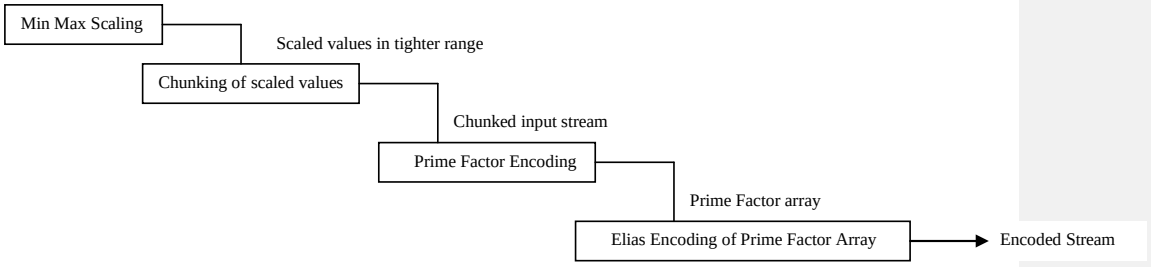


Figure 2: Overview of the Prime Factor Scheme

We exploited the skewed nature of the encoded data by using the Elias variable length coding scheme (Elias, 1975). Elias encoding works by encoding smaller integers in a smaller number of bits, thus further improving the degree of compression.

The next section will describe the details involved in step 3 of the above process, the PFC encoding algorithm.

The Prime Factor Encoding Algorithm

The algorithm takes as its input the scaled set of values produced by the min-max scaling technique. For each chunk, every scaled value is converted into the prime number that is closest to it. We refer to each such prime number as a *prime factor*. The algorithm makes use of two

operators which we define below. Both operators α and β take as their input a pair of prime factors V_k and V_{k+1}.

Definition 1: The operator α is defined by $\alpha(V_k, V_{k+1}) =$ nearest prime $(V_k + V_{k+1} + I(V_{k+1}) - I(V_k))$, where $I(V_{k+1})$, $I(V_k)$ denote the ordinal (index) positions of V_{k+1} and V_k on the prime number scale. The operator takes two primes (V_k, V_{k+1}) adds them together with the difference in index positions between the 2nd prime and the 1st prime and converts the sum obtained to the nearest prime number.
Definition 2: The operator β is defined by $\beta(V_k, V_{k+1}) =$ nearest prime$(V_k + V_{k+1})$

The α operator is applied pair-wise across all values (a total of c prime factors) in the chunk. This yields a stream of c/2 prime factors. The α operator is then applied recursively on the processed stream until a single prime factor is obtained. This recursive procedure gives rise to a tree of height $\log_2 c$ where c is the chunk size. We refer to this tree as the *Prime Index Tree*.
In parallel with the construction of the prime index tree we construct another tree, called the *Prime Tree*. The prime tree is constructed in the same manner as its prime index counterpart except that we apply the β operator, instead of the α operator.

We illustrate the construction of the trees with the help of the following example. For the sake of simplicity we take the cube to be of size 4, the chunk size c to be 4 (i.e. we have only one chunk) and the scale range [nmin, nmax] to be [0, 101].

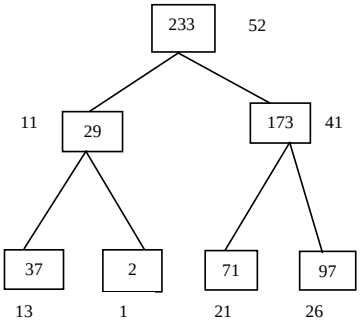


Figure 3a: The construction of the Prime Index Tree

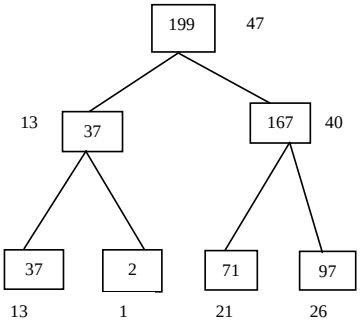


Figure 3b: The construction of the Prime Tree

For the prime index tree (Figure 3a), the prime values 37 and 2 at the leaf level are summed as 37+2+I(2)-I(37), which is transformed to its nearest prime number, 29. Similarly 71 and 97 when processed yield the prime number, 173. The node values 29 and 173 in turn yield a root value of 233.

As shown in Figure 3b, the leaf values 37 and 2 when summed and converted into its nearest prime yields a value of 37. Following the same process, we obtain values of 167 and 199 for the remaining nodes.

We annotate each internal node by its corresponding index value. The encoded array E now consists of the differences in index positions across corresponding positions in the two trees. These differences are referred to as *differentials*. For the example above the array turns out to be E = {5, -2, 1}. We also store the index value of each prime tree root separately in another array R. Each chunk gives rise to its own prime tree root, and for the simple example above, since we have only one chunk, this results in a single value {47} for R.

Decoding requires the use of the two arrays E and R. We start with the first value in array R, which is 47, and then add it to the first value in array E which is 5, yielding the value 52. We use 52 as an index into a table of primes (this table has the pseudo prime 0 added to it with index value 0) and extract 233 as the corresponding prime value. We now search for pairs of primes (P1, P2) which satisfy the condition:
α(P1, P2) = 233, β(P3, P4) = 199, l(P3) = l(P1+2) and l(P4) = l(P2-1) --------(1), where P3, P4 are the corresponding nodes to P1, P2 in the prime tree

This search in general would yield a set S of candidate pairs for (P1, P2) rather than a single pair. In order to extract the correct pair, we associate an integer with each internal node of the Prime Index tree which records the ordinal position within the set S which would enable us to descend to the next level of the tree. In this case this integer turns out to be 0 since there is only one pair that satisfies condition (1). These integers are collectively referred to as *offsets*. The complete set of offsets for the tree above is {0, 0, 0}. The complete version of E contains the sequence of differentials followed by the sequence of offsets. Thus for our example we have E = {5,2,-1,0,0,0}. We are now able to decode by descending both trees in parallel and recover the original set of leaf values 37, 2, 71 and 97.

For ease of understanding, the encoding process above has been described for a 1-d dimensional case. The extension to d dimensions follows naturally by encoding along each dimension in sequence. For example, if we have a 2 dimensional cube $<D_1, D_2>$ we would first construct 2-dimensional chunks. With a chunk size of 16 and the use of equal width across each dimension, each chunk would consist of a 4 by 4 2-d array. We first run the encoding process across dimension D_1. This would yield a 1-dimensional array consisting of 4 prime root values for each chunk. The differentials and offsets that result from this encoding are stored in a temporary cache. The 4 prime root values from encoding on D_1 are then subjected to the encoding process across dimension D_2 to yield the final encoding. The differentials and offsets that result from encoding along D_2 are merged with those from encoding along dimension D_1 to yield the final set of encoded values.

Encoding Performance

The encoding takes place in four steps as given in Figure 2. Steps 1 and 2, involving scaling and chunking can be done together. As data is read it can be scaled on the fly with the chunking process. If the original data is stored in dimension order (D_1, D_2, ……D_d) with the rightmost indices changing more rapidly, then it can be shown the I/O complexity involved in chunking is $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$, where M is the available memory size and B is the block size of the underlying database system. The reader is referred to [Vitter 1999] for a proof. The I/O complexity of steps 3 and 4 is $O\left(\frac{N}{B}\right)$. Thus the I/O complexity is bounded by the $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ term required for the chunking step.

However it should be noted that this only reflects a one time cost in reorganizing the data from dimension order to chunked format. Once this is done, no further chunking is required as updates to values do not affect the chunk structure. Thus the time complexity on a regular basis would be bounded above by $O\left(\frac{N}{B}\right)$. The only exception occurs when the dimensions are reorganized and either grow or shrink as a result. This would require repetition of the chunking step.

The Rationale behind the Prime Factor Scheme

Prime numbers provide us with a natural way of constraining the search space since they are much less dense than ordinary integers. The first 100 positive integers are distributed across only 25 primes. At the same time the primes themselves become less dense as we move up the integer scale (Andrews, 1994). The next 900 positive integers after 100 only contain 143 primes. This means that the prime number encoding technique has good scalability with respect to data value size. From the error point of view the use of primes introduces only small errors as it is possible to find a prime in close neighborhood to any given integer (Andrews, 1994). Theorem 1 below gives the distribution of primes in the general case.

Theorem 1

The number of prime numbers less than or equal to a given number N approaches $\frac{N}{\log_e(N)}$ for large N.

Proof: The reader is referred to (Andrews, 1994) for a proof.

Theorem 1 reinforces the observations made above. Firstly, the division by the logarithm term ensures that the primes are less dense that ordinary integers. Secondly, the average gap between a given prime N and its successor is approximately $N / \left(\frac{N}{\log_e N} \right) = \log_e N$. This means that encoding a number N using its nearest prime will result in an absolute error of $\frac{1}{2}\log_e N$ on the average. These properties make prime numbers an attractive building block for encoding numerical data.

The basic idea that we utilize is to convert a stream of primes into a single prime, the Prime Tree root value by a series of pair wise add operations. We then augment the root value with a set of coefficients to enable us to decode. The use of prime numbers enables us to drastically reduce the search space involved in decoding and as a consequence reduce the space required to store the encoded data. As an example consider the prime root value of 29. In order to decode (assuming that we simply use the Prime Tree) to the next level we have to consider just eight combinations (5,23), (23,5), (7,23), (23,7), (11,19), (19,11) , (13, 17) and (17,13). On the other hand if prime numbers were not used as the basis, then we would have a total of 30 combinations to consider – in general, if N were the prime root, then N+1 combinations would have to be tested.

The use of the Prime Index tree in conjunction with the Prime Tree enables us to constrain the search space even further. With the introduction of the former we are able to distinguish between pairs such as (5,23) and (23,5). The pair (5,23) encodes as 31 in the Prime Index tree and 29 in the Prime tree, whereas (23,5) encodes as 23 in the Prime Index tree and 29 in the Prime tree. Apart from this, the other major benefit of growing two trees instead of one is that we can encode taking the differentials between corresponding nodes across the two trees rather than node values themselves. Since the two trees evolve from a common set of leaf values, the α and β operators evaluate to approximately the same values which in turn causes the differentials to be much smaller than the node values themselves (see Figure 13a in the “Experimental Setup” section).

Comparison of the Wavelet and Prime Factor Schemes

The wavelet and prime factor schemes both use the concept of reducing the original data to differentials between progressively increasing sub sets. However one major difference is that that the Wavelet scheme uses thresholding to drop some of the differentials. As noted before, this makes it unstable with respect to the error rate (Garofalakis and Gibbons, 2004) and this is confirmed by our results which we present later. In contrast, the Prime Factor scheme does not use thresholding but exploits the skew induced by the prime factor transformation to encode the resulting coefficients using an Elias code. This means that *every* value in the original data set is represented in the encoded version which results in much greater stability over the Wavelet scheme.

Another major difference is that the encoded data in the Wavelet scheme are represented as relational tuples in the form $<i_1, i_2, \ldots i_d, c>$ where each i_k (in the range 1 to d) identifies the index position within dimension k in which the surviving coefficient with value c is located. For large values of d, the degree of compression obtained can degrade quite severely as each index takes up additional storage. In contrast, the Prime Factor scheme does not require any such indexes as thresholding is not used and thus we would expect it to have relatively better compression for high dimensionality data.

The Prime Factor scheme also copes well in the case of sparse data. All zero values in the original data will scale to 0 with the scaling scheme that we use. Let us consider an example where we have a run of six zero's followed by the primes 101 and 61 in the scaled data set and suppose that we use a chunk size of 8. Since we have extended the prime number set to include the pseudo prime 0, the 0 values encode to 0 and this results in a *left prefix* tree where we have non-zero values preceded by a run of zero values. Figures 4a and 4b below show the resulting trees.

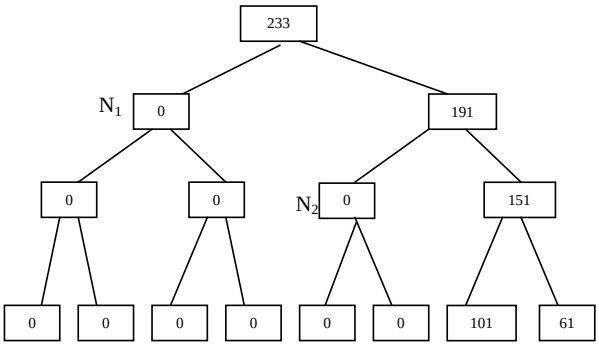


Figure 4a: Left Prefix Prime Index Tree

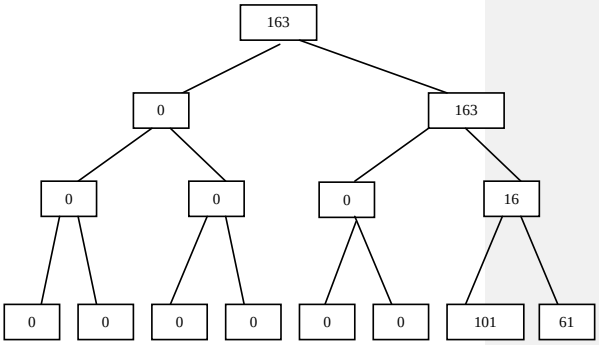


Figure 4b: Left Prefix Prime Tree

Similarly, when the original data stream consists of a sequence of non zero values followed by a run of zero values, we obtain a *right prefix* tree after encoding, where a non zero internal node has a zero valued node as its sibling in each of the Prime Index and Prime trees.

The left and right prefix trees can be "collapsed" and stored compactly as indicated by the following Lemma and Theorem 2.

Lemma 1 *If* $\beta(P_1, P_2) = 0$, *then we must have* $P_1+P_2=0$.

Proof: From the definition of β, we have $\beta(P_1, P_2) = P_1+P_2+s = 0$ -------(1), where s is an integer that rounds the sum of P_1+P_2 to the nearest prime factor. Suppose that $s \neq 0$. We now consider two cases, s>0 and s<0.

Case 1 s<0. Since s<0, it follows that $P_1+P_2<1$------(2), otherwise, it will be impossible to round the sum of P_1+P_2 to 0. We also have $P_1+P_2 \geq 0$ -----(3) since $P_1 \geq 0$, $P_2 \geq 0$. From (2) and (3) it follows that $P_1+P_2=0$.

Case 2 s>0. The proof is similar to Case 1 above.

Since s >0, the only way that we can satisfy (1) is for $P_1+P_2 \leq 0$ ---(4), in order to have any chance of rounding to 0.

From (3) and (4) it follows that $P_1+P_2=0$.

Theorem 2 *If in a left prefix or right prefix tree a given node encodes to zero, then it follows that the entire sub tree under that node and its associated leaf values must be zero valued.*

Proof: We use the proof by contradiction method as a proof sketch. We start with a pair of leaf nodes having values P_1 and P_2. Suppose that $P_1 \neq 0$, $P_2 \neq 0$. Foe the prime index tree, we have $\alpha(P_1, P_2) = P_1+P_2+I(P_2)-I(P_1)+r =0$ ---------(5), where r is an integer that rounds the value of the sum of the preceding terms to the nearest prime factor.

From Lemma 1 we have $P_1+P_2=0$. Substituting this in (5), we have $\alpha(P_1, P_2) = I(P_2)-I(P_1)+r =0$. Thus $I(P_1)=I(P_2)+r$. Thus, if r>0, we have $I(P_1)>I(P_2)$, which in turn means that $P_1>P_2$. With $P_1>P_2$ and $P_1, P_2 \geq 0$, we have $P_1+P_2>0$, which leads to a contradiction. Similarly, with r<0, we have $P_2>P_1$, which again leads to $P_1+P_2>0$. Thus, in either case, our original assumption of $P_1 \neq 0$, $P_2 \neq 0$ must be in error and we have $P_1=0$ and $P_2=0$. A similar proof holds for the prime tree.

This means that any node with a value of 0 must give rise to a pair of zero valued child nodes (this is true of both the Prime Index and Prime trees). Each of these zero valued child nodes in turn will yield children who have zero valued. This completes the proof sketch.

We now return to the example in Figures 4a and 4b. In Figure 4a we can collapse the tree by pruning all branches for the sub trees rooted at nodes N_1 and N_2. This means that we only need to store a total of six coefficients, made up of 3 differentials and 3 offsets (as opposed to a total of 14 for the non-sparse case).

Thus it can be seen that the Prime Factor scheme adapts to a sparse data set without the need to keep explicit indexes to keep track of the zero values in the data set.

Encoding Error Rate for the Prime Factor Algorithm

In this section we present a formal analysis for the average relative error involved in decoding with the Prime Factor scheme. Theorem 2 below quantifies this error rate.

Theorem 3 : The average relative error is approximately $\frac{1}{2}\frac{\log_e(C\overline{V})}{\overline{V}}$, where C is the chunk size, and $\overline{V}$ is the average value of an element within a chunk

Proof Sketch:

Consider the Prime tree given in Figure 5 below with chunk size 8.

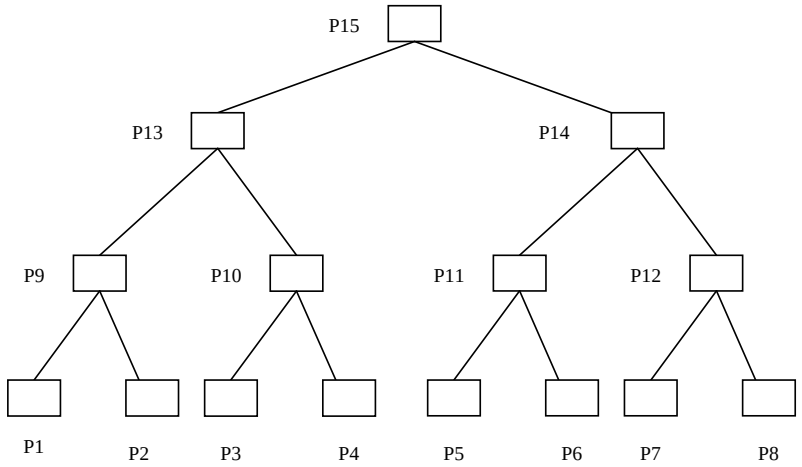


Figure 5: Error Tree for Chunk Size 8

The value of an internal node, say P_9 is given by $P_9 = P_1 + P_2 + \Delta_{P_2, P_1}$
where Δ is an error term that denotes the approximation to the nearest prime number
Similarly, $P_{10} = P_3 + P_4 + \Delta_{P_4, P_3}$ and $P_{13} = P_9 + P_{10} + \Delta_{P_{10}, P_9}$.

Substituting for P_9 and P_{10} in the expression for P_{13} we obtain

$$P_{13} = P_1 + P_2 + P_3 + P_4 + \Delta_{P_2, P_1} + \Delta_{P_4, P_3} + \Delta_{P_{10}, P_9}$$

Thus it can be seen that a parent node's value is given by the sum of the values of the leaf node values that can be reached by the parent node plus the error terms at the intermediate node levels.

In general, we have P_{2C-1}, the root node value given by $P_{2C-1} = \sum_{i=1}^{C} P_i + \sum_{i=1}^{C-1} \Delta_{P_{2i, 2i-1}}$

The sum of the values T in the chunk is given by $T = \sum_{i=1}^{C} P_i$

Thus we have $d = P_{2C-1} - T = \sum_{i=1}^{C-1} \Delta_{P_{2i, 2i-1}}$ ----- (1)

From Theorem 1, we have the average gap between a prime number P and its next as $\log_e(P)$. This result can be used to estimate the average value of the error coefficient Δ as $\Delta \approx \frac{\log_e(P_{LC} + P_{RC})}{2}$, where P_{LC}, P_{RC} represent the left and right children of a given parent node, the divison by 2 is necessary because the Prime Factor algorithm encodes each value with its nearest prime number.

Put $P_{2i} + P_{2i-1} = K_i T$, where $K_i \leq 1$ for i = 1, 2, …., C -1

Each error term can now be written as $\Delta_{P_{2i,2i-1}} = \frac{1}{2}\log_e(K_i T)$

Thus from (1) above we have $d \approx \frac{1}{2}\sum_{i=1}^{C-1}\log_e(K_i T) \approx \frac{1}{2}\sum_{i=1}^{C-1}\log_e(K_i) + \frac{1}{2}(C-1)\log_e(T)$

$$\approx \frac{1}{2}\sum_{i=1}^{C-1}(K_i - 1) + \frac{1}{2}(C-1)\log_e(T)$$

using the Taylor series expansion for the log function and taking a first order approximation for $0 \leq K_i \leq 2$.

From the definition of K_i, we have $\sum_{i=1}^{C-1} K_i \approx \log_2(C)$

Since $\log_2(C) << \frac{1}{2}(C-1) << (C-1)\log_e(T)$, we have $d \approx \frac{1}{2}(C-1)\log_e(T)$

The relative error, $\frac{d}{C\overline{V}} \approx \frac{1}{2}\frac{(C-1)}{C}\frac{\log_e(C\overline{V})}{\overline{V}}$. Note : the scaling terms to scale the difference d and the sum $C\overline{V}$ to the original data scale do not need to be considered as they cancel each other off in the numerator and denominator.

$$\text{Thus } d \approx \frac{1}{2}\frac{\log_e(C\overline{V})}{\overline{V}} \text{ for large C}$$

The above result captures the relationship between the error rate, chunk size and scale range. As the chunk size increases, the error rate grows as a log function. This is shown visually in Figure 11b, whereby the average error rate increases sub linearly with respect to increasing chunk size. The above expression also shows that for a given chunk size C, the error rate decreases with an increase in scale range size. As the scale range size increases, the $\overline{V}$ term increases at a faster rate than the $\log_e(C\overline{V})$ term, thus causing a drop in the error rate. Figure 10b displays this trend.

ON LINE RECONSTRUCTION OF QUERIES

In this section we examine how multi-dimensional range sum queries are answered with the Prime Factor scheme. These queries are of the form:

$\{l_i \leq D_i \leq h_i | \; l_i, h_i \in \{0,1,\ldots\ldots, \; | \; D_i \; | -1\}, h_i = l_i + \Delta_i\}$, where Δ_i is a positive integer.

We will first derive an expression for the time complexity involved in answering a 1-dimensional query as some of the concepts involved are shared with the general the n-dimensional case.

Answering One Dimensional Range Sum Queries

In this case we need to consider three regions R_1, R_2 and R_3 as shown in Figure 6 below. The query requires the sum of all elements in the 1-dimensional cube which are in the range $[l_1, h_1]$, with l_1= 22 and h_1=690. The bulk of the query resides in region R_2 (the shaded region) which is aligned with the chunk structure. The sum across this region can be answered very efficiently by taking the sum of the root values for the chunks that span this region, thus avoiding the need for decoding across a large portion of the query. Since each of these values corresponds to the root associated with the Prime tree they are a close approximation to the sum contained within a chunk.

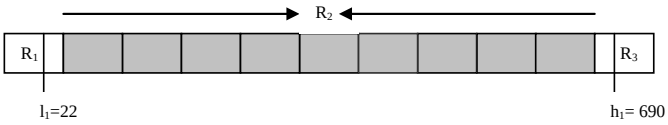


Figure 6 Decoding and answering a 1 dimensional query

Furthermore, with a large enough chunk size we would expect that the root values comprise a very small fraction of the size of the original cube and thus be either held in memory or stored on a small number of blocks on secondary storage (with a chunk size of 256 for example, these root values comprise just 1/256 of the number of elements within the original cube). Note that this is true irrespective of the dimensionality of the query or that of the cube.

The only decoding that is necessary is for the two chunks at the "ends" of the query, i.e. regions R_1 and R_3. Thus it can be seen that the chunking has helped to minimize the effort involved in decoding. It is also important to note that the decoding effort is dependent on the dimensionality of the query involved and not on the dimensionality of the cube. For example, for a query such as $[l_3, h_3]$ (say with l_3 = 22 and h_3 = 690) on dimension 3 of a 5-dimensional cube, the number of chunks to be decoded is still 2 since we only need to consider the one dimensional slice along dimension 3 which is aggregated across dimensions 1, 2, 4 and 5. Again, this observation holds for the general case as well. However, the amount of effort (in terms of the number of chunks to be decoded) involved will depend on the dimensionality of the query as we shall see in the next section.

In the next section we will present an expression for the general case, involving queries on n dimensions.

The n-Dimensional Case

Suppose that the query spans n dimensions. On each of these dimensions we can define three regions, the two regions at the ends of the query and the "middle" region which is aligned with the chunk structure. This leads to a total of 3^n regions. Out of these the majority of the query is aligned with the chunk structure in a single contiguous region in n-dimensional space, thus requiring decoding across $3^n - 1$ regions. Each of these regions is on a boundary of the query in n-dimensional space and thus tends to be small in size. Theorem 4 below quantifies the amount of decoding effort involved in the general case.

Theorem 4 *The total number of chunks to be decoded for an n-dimensional query is* $\left(\frac{3^n - 2^n - 1}{n}\right)\sum_{i=1}^{n}(d_i - 2) + 2^n$ where $d_i = \left\lceil \frac{\Delta_i}{c} \right\rceil$

Proof Sketch: We will first consider the regions on the "corners" of the cube defined by the query. There are 2^n such corners. Each of these corner regions are contained in exactly one chunk. This leaves a total of $(3^n - 2^n - 1)$ regions, occurring along n dimensions. Each of these regions along dimension i will span $(d_i - 2)$ chunks, since 2 of the total number comprises the corner chunks. Thus the total number of these chunks is $\left(\frac{3^n - 2^n - 1}{n}\right)\sum_{i=1}^{n}(d_i - 2)$. To this number we must add the 2^n corner regions. This completes the proof.

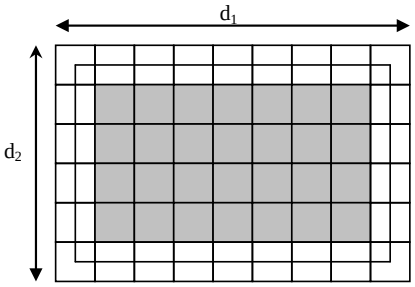


Figure 7a: The Region aligned with the chunk structure

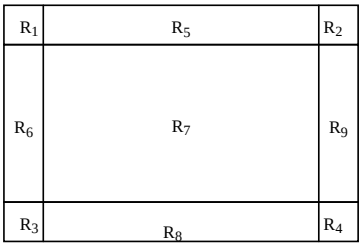


Figure 7b: Regions bounded by a 2-dimensional query

Figures 7a and 7b provide a geometrical explanation for the 2D case. Figure 7a illustrates the region that requires no decoding, and it can be seen that it covers the bulk of the query space. Figure 7b gives a complete breakdown of regions bounded by the query. The region R_7 corresponds to the shaded region in Figure 7a; R_1, R_2, R_3, R_4 represent the four corner regions that each require decoding a single chunk; the regions R_5, R_6, R_8 and R_9 in general span larger regions containing multiple chunks that require decoding.

Theorem 5 *For a given query dimensionality n, the chunking scheme improves decoding efficiency by a minimum factor of* $\left(\frac{n}{3^n - 2^n - 1}\right)\frac{\prod_{i=1}^{n} d_i}{\sum_{i=1}^{n} d_i}$

Proof Sketch: Follows from Theorem 4 above and the fact that an n-dimensional query requires decoding $\prod_{i=1}^{n} d_i$ chunks if aggregate sums (prime tree root values) are not stored at chunk level.

IMPLEMENTATION CONSIDERATIONS

In this section we will provide a brief overview of the implementation of some of the major components of PFC. The system was implemented entirely in Java. We made use of a table of prime numbers (Alfeld) readily available from the Web. All other functionality was custom built.

Encoding

The three main functions that were used extensively with respect to encoding were Find_Nearest_Prime(N), Evaluate_Alpha(N, M) and Evaluate_Beta(N, M). The Find_Nearest_Prime(N) function was fairly straightforward to implement as the use of the prime table meant that the complexity of testing for primes was avoided. The other two functions,

Evaluate_Alpha(N, M) and Evaluate_Beta(N, M), formed the core of the PFC scheme and were used to build the Prime Index and Prime Trees respectively. These functions essentially took two prime numbers as their arguments and converted their sum to the nearest prime number (the Evaluate_Alpha function also added a component that involved the difference in index positions between the two given primes). Thus these two functions were quite straightforward to implement as well, as they only involved simple computations.

Decoding

With respect to decoding there were two cases to consider. The first case involved chunks that required no recovery of the individual leaf nodes within the chunk. This case covers a large fraction of the chunks involved in a query, as quantified by Theorem 4. For these chunks all that was required was extraction of the prime tree root value and subsequent re-scaling to transform the value back to the original source data scale range.

The second case involved chunks that required actual decoding and this was accomplished through a function, Descend_Prime_Tree(N), that was used to descend one level down the Prime Tree. Basically, this function took a prime number denoting the ancestor node in the corresponding tree as its argument, and then returned a set containing pairs of primes that were candidates for the child nodes. We used a heuristic to speed up the generation of these candidate sets. For a given ancestor node, with prime value N, we extracted the prime number M that is nearest to $\lceil N/2 \rceil$. The pair (M, M) is guaranteed to be a candidate for the Prime Tree child nodes. This pair was used as a starting point for the generation of the rest of the pairs. Figure 8 below illustrates the algorithm used for generating the Prime Tree candidate set.

```
Descend_Prime_Tree (N)
{
    C={};
    M = nearest_prime (⌈N/2⌉);
    C = C ∪ {(M, M)};
     Indx = I(M);                          // the index value of prime M
     U = M;
     L = nearest_prime(P[Indx-1]);         // the largest prime less than M
     while (L ≤ nmin and U ≥ nmax)         // [nmin..nmax] represents the scale range
     {
        V = β(U, L);
        if (V = = N) then
        {
            C = C ∪ {(L, U), (U, L)};
            U = nearest_prime(P[I[U]+1]);  // increase upper value in prime pair
         }
         else if (V <N) then
            U = nearest_prime(P[I[U]+1]);   // increase upper value in prime pair
         else L = nearest_prime(P[I[L]-1]);         // decrease lower value in prime pair
     }
}
```

Figure 8: Algorithm for generating Candidate Prime Pairs

In order to further speed up the process of decoding we cached the prime pairs for a given integer N. The decoding process benefits from such a cache as prime tree nodes with a given value N are likely to recur many times across different chunks. In such cases, the process of generating prime pairs is reduced to a fast in-memory table lookup. Our experiments show that this cache occupied less than 1% of the storage of the original source data.

With the compilation of the candidate prime pairs, the decoding process reduces to applying the differential and offset coefficients to descend both trees in parallel to extract the leaf level values, as described earlier in the section on “The Prime Factor Encoding Algorithm”.

EXPERIMENTAL SETUP

In this section we describe the experiments we carried out with the Prime Factor and Wavelet schemes. The experimentation could be broadly divided into two main categories:

- A performance comparison between the two schemes,
- An investigation into the performance of the PFC scheme with respect to key parameters

We use two metrics, *Compression Ratio,* and *Relative Error* to quantify the degree of compression and error respectively. The compression ratio (cr) is defined by: cr = encoded data size in bytes/size of original data set in bytes, while the relative error (re) is given by: re = |S-R|/S, where S represents sum of the query on the original (un-encoded) data set, and R is the reconstructed sum after decoding has been carried out with the Prime Factor or Wavelet schemes.

Comparison of the Prime Factor and Wavelet Schemes

In this section we focus on a performance comparison between the Prime Factor and Wavelet schemes on a range of different data sets. We used both real-world data from the (US Census) that was used in (Vitter *et al*, 1998; 1999, Chakrabarti *et al*, 2000) as well as synthetic data sets for this purpose.

The synthetic data sets were modeled on the criteria used in (Ng and Ravishankar, 1997). Two parameters, degree of skew and degree of variation were used in their generation. The degree of skew was taken to be *high* when 70 percent of the data elements were drawn from 30% of the domain value range, with the other 30% of the data drawn from a uniform data distribution in the domain value range. On the other hand, the degree of variation was taken to be *high* when the difference in data element values is more than 100% of the average data element value. When the differences in data element values was no more than 10% of the average data element value, then the data variation was taken to be *low*. Variation in the value of these two parameters gave rise to four different data sets: (No Skew, Low Variance), (High Skew Low Variance), (No Skew, High Variance) and (High Skew, High Variance).

Decoding Time

The first experiment was run on the US Census data and was designed to gain an insight into how the decoding times varied with the size (i.e. the portion of the data that the query retrieved, expressed as a percentage) of the query posed. We used a chunk size of 64 and a scale range of [0..307] for the PFC and a threshold of 7 % for the Wavelet. These parameter settings ensured that the two schemes produced roughly the same degree of compression in order to isolate the effect of encoded file size on the timings. At each value of query size we ran 50 tests, consisting of a batch of 10 for each value of the query dimensionality parameter which ranged from 1 to 5, and the average time was recorded across the 50 runs.

Figure 9 shows that the Prime Factor outperformed the Wavelet scheme in the entire range tested. The two curves start at roughly the same point at the lower end of the size scale, but diverge quite

rapidly as the query size increases. The good performance of the Prime Factor scheme can be attributed to two factors.

Firstly, its lower average (taken across the entire range of query size) decoding time per data element was 0.0021 ms versus 0.0057 ms for the Wavelet. In the case of the Prime Factor scheme the information (i.e. the decoding coefficients) necessary for decoding a data element is localized within the chunk that encapsulates it. There is no concept of a single global tree, since each chunk is encoded separately, thus giving rise to a collection of independent trees. This means that chunks can be decoded independently from each other, and only those coefficients belonging to chunks that require decoding need to be examined. On the other hand, the Wavelet scheme encodes using a single tree and thus query processing involves searching through the entire set of wavelet coefficients to determine the contributions made by each individual coefficient (Vitter, 1999).

Secondly, PFC decodes a much smaller number of data elements, as only chunks along the boundaries of a query need to be decoded (as proved by Theorem 5).

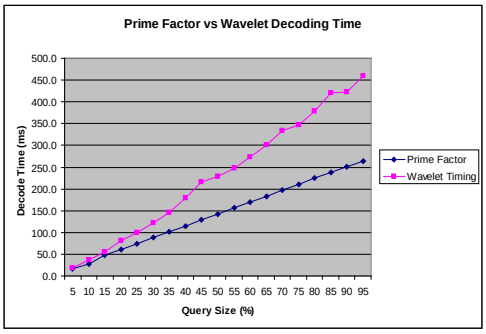


Figure 9: Effect of Query Size on Decoding Time

Error Rates

In the next set of experiments we compared the two schemes on error rate. As before we varied the query size in steps of 5 in the range 5% to 90% and measured the relative error at each step. For each size value we ran 10 tests, with each test retrieving data from different regions in the data set. For each batch of 10 runs we measured the minimum, maximum and average relative error values (all expressed as percentages) for the PFC (with scale range of [0..307]) and the Wavelet (with thresholding set at 7%).

As can be seen from Tables 1 and 2 for roughly the same compression ratio the two schemes have significantly different error throughout the range tested. The PFC scheme exhibits a small average error rate of around 0.3% right up to the 90% mark (this was true for higher percentages of the query size parameter - up to 99% which we have omitted for reasons of brevity). As Theorem 2 demonstrates the average error rate for the PFC is essentially dependent on the scale range and the chunk size used and is not a function of query size.

The differential between the average error rates for the two schemes is much higher at the lower end of the query size range (e.g. from 5% to 55%) than at the upper end. At the upper end of the query size range the Wavelet's performance improves as the top level wavelet coefficient representing the overall mean of the data set assumes more importance in reconstruction.

The relative error for the PFC scheme is remarkably stable throughout the query size range. The minimum, average and maximum values are much closer together than its Wavelet counterparts. With the Wavelet scheme we have significant deviations between the minimum and maximum error values in the entire size range. This is in line with previous research (Vitter *et al*, 1999), (Garofalakis *et al*, 2004). In practice error stability is important as this would inspire more confidence in users of the accuracy of results to ad-hoc OLAP queries.

We cross checked these results with the synthetic data sets that we generated and observed the same trends.

	Compression Ratio
PF 307	8.05
Wavelet (7%)	7.37

Table 1: Comparative Compression Rates

Query Size (%)	**Relative Error (%)**					
	Min PFC 307	**Min Wavelet**	**Avg PFC 307**	**Avg Wavelet**	**Max PF 307**	**Max Wavelet**
5%	0.104	0.479	0.292	6.046	0.453	15.446
10%	0.151	0.181	0.280	4.738	0.424	16.541
15%	0.172	0.012	0.345	1.932	0.438	5.063
20%	0.144	0.136	0.291	2.210	0.418	5.868
25%	0.167	0.483	0.317	2.994	0.410	5.568
30%	0.211	0.282	0.328	2.275	0.454	4.993
35%	0.230	0.706	0.341	2.785	0.390	5.672
40%	0.249	0.074	0.326	0.934	0.371	5.071
45%	0.263	0.011	0.347	0.728	0.402	1.168
50%	0.286	0.195	0.338	0.559	0.394	1.539
55%	0.326	0.001	0.361	0.535	0.386	1.102
60%	0.305	0.046	0.342	0.351	0.402	0.932
65%	0.311	0.327	0.332	0.524	0.360	0.958
70%	0.315	0.214	0.334	0.635	0.380	1.006
75%	0.321	0.067	0.356	0.515	0.382	0.877
80%	0.317	0.041	0.339	0.382	0.372	0.713
85%	0.314	0.045	0.328	0.349	0.343	0.759
90%	0.320	0.007	0.329	0.206	0.342	0.507

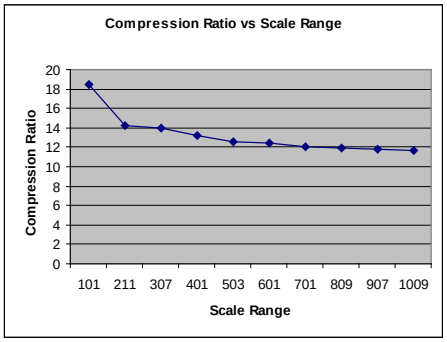
Table 2: Comparative error rates

Sensitivity of Prime Factor Performance on Key Parameters

We investigated the effects of scale range size and chunk size on performance. We also looked at the distributions of the encoded coefficients in order to gain an insight into whether the Elias code would be an effective representation mechanism. The experiments were conducted on the real world US Census data set.

Effect of Scale Range Size

In this set of experiments we tested the effect of scale range on both compression ratio and accuracy. We kept the lower bound of the scale at 0, and varied the upper bound from 101 to 1009 in approximate steps of 100. Figures 10a and 10b track the effects of this variation on compression ratio and relative error. We measured the relative error on queries that randomly picked a 20% sample of the data (in actual fact the relative error was obtained by averaging across 10 runs for each value of the scale range parameter).



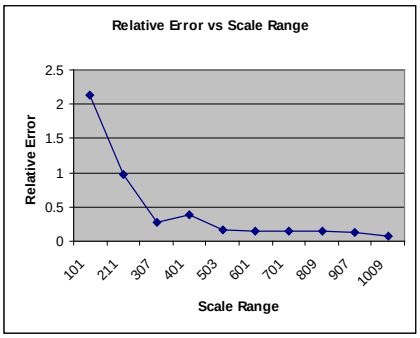


Figure 10a: Effects on Compression Ratio

Figure 10b: Effects on Relative Error

As shown in Figure 10a the compression ratio decreases as the scale range widens. It decreases steadily from a maximum of 18.4 at the narrow range of [0..101] and tends to flatten out at wider ranges (i.e. at [0..701] and beyond). At low scale ranges values the data tends to have a low degree of variation in value and this in turn induces a large number of zero or near zero coefficients after encoding. Widening of the scale produces the opposite effect, resulting in relatively low compression.

We would also expect a trade-off between the degree of compression and accuracy. At higher compression ratios the error rate would tend to be higher than at lower degrees of compression. Figure 10b exhibits this trade-off. As the scale widens we see a steady drop in relative error until the scale range value of [0..503] is reached, and thereafter the error continues to drop but at a much lower rate. At the narrow scale ranges the prime factors obtained by rounding off the scaled values tends to produce a much coarser approximation than at the wider scale ranges, and this has the effect of inflating the error at the lower scale ranges.

Effect of Chunk Size

The total size of the root values produced by the chunks is also of interest as this affects the efficiency of decoding. As we saw in earlier in the section on "On Line Reconstruction of Queries" a large portion of the query can be answered by simply summing up the root values produced by the chunks. A smaller root file (used to store the root values of the chunks) size would thus greatly speed up the decoding process. We thus experimented with differing chunk sizes to test the effect on the root file size. We would expect the root file size to decrease monotonically with an increase in chunk size. Figure 11a below confirms that this is the case. However, we also wanted to test whether or not larger chunk sizes have an adverse effect on accuracy. Figure 11b below shows that the relative error decreases slightly in the range 256 to 64 and then decreases more rapidly thereafter. Thus we have a basic trade-off between root file size and accuracy. Figure 11b shows that the decrease in the error is not significant until we reach a chunk size of 16 or smaller. Thus from a practical viewpoint a larger chunk size of 256 or 128 would yield a very small root file size (approximately 0.32% of the original data set size for a chunk size of 256) and a reasonable level of accuracy. We repeated this experiment on different data sets and found a similar trend.

We would thus expect that in most cases that the root file would be small enough to be cached in memory when using a chunk size of 256 or greater.

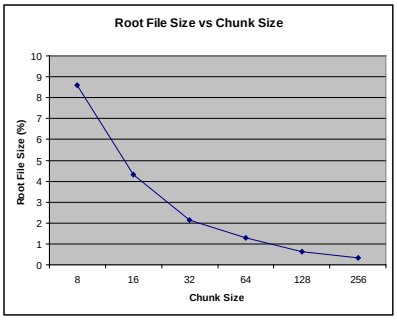


Figure 11a: Effect of Chunk Size on Root File Size

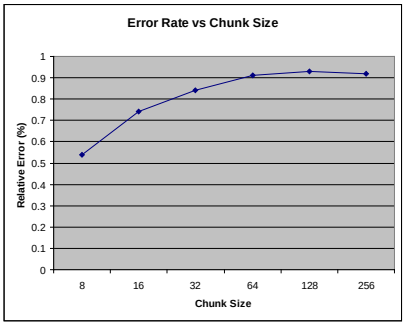


Figure 11b: Effect of Chunk Size on Relative Error

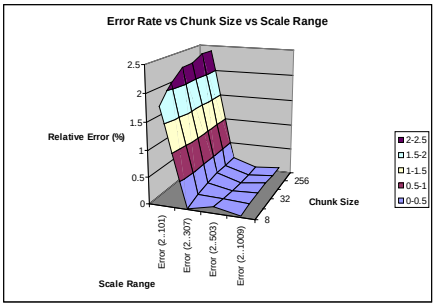


Figure 12a: Simultaneous Effects of Chunk Size and Scale Range on Relative Error (Census Data Set)

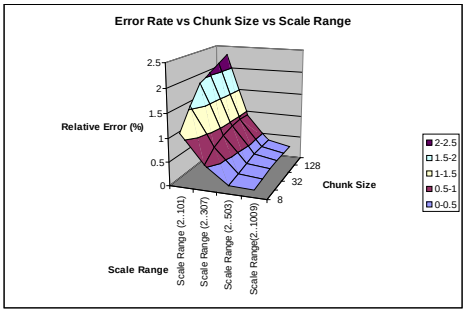


Figure 12b: Simultaneous Effects of Chunk Size and Scale Range on Relative Error (High Variance High Skew Data Set)

Having established the effects of the individual effects of chunk size and scale range on the error rate, we next investigated the simultaneous effects of both these parameters on the relative error. The 3D surface plot in Figure 12a clearly illustrates two facts: firstly, the error rate is sensitive to both scale range and chunk size (in accordance with the results presented in Figures 10b and 11b); secondly, the two parameters interact with each other. With larger values of scale range, for example (2..1009), variation in chunk size produces an increase of only 0.05% in the relative error rate, whereas the corresponding increase at the low end of the scale range, (2..101) was 0.6%. Figure 12b shows similar trends for the (High Variance, High Skew) synthetic data set. Results for the other three synthetic data sets had similar behavior.

Effect of Elias Coding on Prime Factor Performance

We next conducted an experiment to test the effect of Elias coding on the coefficients produced by the PFC scheme. We plotted a histogram of the distributions of the differential (we plotted the absolute values, as a differential can take a negative value) and *offset* coefficients to test whether these would be skewed. A high degree of skew would mean that the Elias code would be able to perform better by encoding the most frequent symbols in the smallest number of bits thus improving the compression ratio.

Figures 13a and 13b show that there is a high degree of left skew, i.e. the coefficients tend to be clustered around the smaller values. These results show that a good variable length encoder such as the Elias coder would be successful in further compressing the raw output of the Prime Factor scheme (i.e. the *differential* and *offset* coefficients). This helps to explain the good compression ratios that we obtain from the Prime Factor scheme (see Figure 10a).

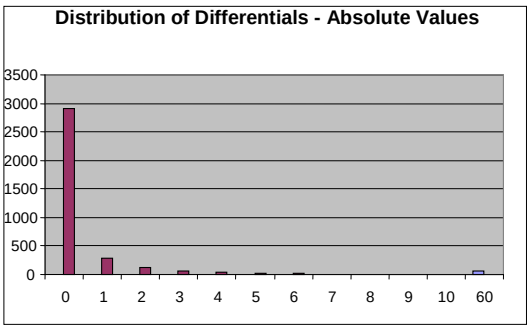


Figure 13a: Distribution of Differentials

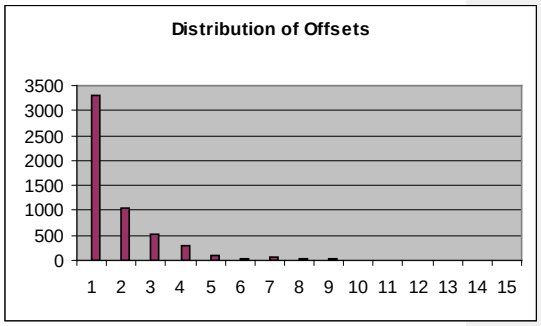


Figure 13b: Distribution of Offsets

Experimentation on Synthetic Data

In order to gain an insight into how the Prime Factor scheme performs with different types of data sets, we generated synthetic data with varying degrees of skew and variability in the data. We adopted the framework provided in (Ng and Ravishankar, 1997) to quantify the skew and data variability parameters as follows:

- The data variance is low when the data is spread so that the values are clustered at a distance no more than 10% from the mean value. It is high when the spread is more than 100% from the mean.
- The distribution of values is skewed when 70% of the values were drawn from 30% of the value domain. When values were drawn randomly from the value domain, then no skew was registered.

These two parameters lead to the generation of four types of data sets: Low Variance and no skew, Low Variance with High Skew, High Variance with no skew and finally High Variance with Skew. We ran the Prime Factor and Wavelet schemes on each of these four data sets and measured the compression ratio. For the Prime Factor scheme we used three different scale range settings: [0..101] (i.e. PF 101), [0..307] (PF 307) and [0..1009] (PF 1009).

	Low Variance	Low Variance Skew	High Variance	High Variance Skew
PF 101	12.47	17.30	7.77	7.53
PF 307	10.96	12.32	6.45	6.19
PF 1009	8.04	9.76	5.15	4.73
Wavelet	7.49	7.52	7.00	6.73

Table 3 Comparative Compression Ratios

Table 3 above shows that for the synthetic data, the relationship between width of scale range and compression ratio mirrors what was obtained with the real-world data. The results with respect to respect to Relative Error also exhibit the same trend as with the real-world data with the Prime Factor for scale ranges [0..307] and [0..1009] performing better than the Wavelet for query sizes up to 80%, and maintaining an error rate of less than 0.1% thereafter.

Table 3 also shows other interesting trends. First of all, we can see the effects of data variability on compression. Across both schemes it is clear that a low degree of variability allows for better

compression performance. This is consistent across all variants of the Prime Factor scheme as well as the Wavelet scheme. At the same time skewness in the data impacts on compression performance, especially with the Prime Factor scheme. In the case of low variability in the data a high degree of skew has a positive effect on compression performance whereas with a higher degree of data variability skew seemed to have the opposite effect. These results are in line with those reported by Ng and Ravishankar.

With a low degree of data variability, the data elements are clustered in value space and leads to a higher proportion of encoded coefficients with smaller values which helps both the Prime Factor and Wavelet schemes. The degree of data skew now has a positive effect on the Prime Factor compression performance. The skew amplifies the clustering effect as a large percentage of data elements occur in a small value domain thus making the encoded coefficients smaller in value. On the other hand when the data variability is high the data elements are spread far apart in value space and thus the skew does not increase the inherent level of value clustering in the source data.

FUTURE TRENDS

One of the main issues facing contemporary Database Systems in general and Data Warehouses in particular, is performance. Given that data storage volumes are increasing rapidly and that much of this information is being incorporated into Data Warehouses, it becomes incumbent that an optimization strategy that scales well with storage volume is put in place. Data compression is one of the most important tools that have been used to combat this problem of increasing storage.

In parallel with this we foresee an increasing research effort directed towards answering multi-dimensional queries more efficiently with the use of novel indexing schemes that are tailored to queries that are typically expressed in data warehouses (Albrecht et al 2000). At the same time we also see a sustained interest in caching synopses of multi-dimensional queries (Shim et al, 2004, Park and Lee, 2005, Gemulla et al 2007) so that parts of the cache can be re-used across several such queries. We see the work presented in this chapter as complementing the research effort in such query optimization strategies. Albrecht el al report that cost reductions of up to 60% was obtained with the use of the cache. Data Compression can be used to leverage the benefits of caching synopses of data by reducing the storage size, thereby improving the cost reduction efficiency. The same holds true for indexing structures such as bitmap indexes. Once again, compression can be applied to further improve the cost benefits of using bitmap indexes for aggregate queries.

CONCLUSION

We have demonstrated the effectiveness of the Prime Factor Compression (PFC) scheme in answering OLAP queries against a data warehouse. In this chapter we presented a detailed design of the PFC scheme, and formulated analytic proofs of several aspects of its performance, including optimizations suitable for use with sparse data. We also presented an implementation strategy for encoding and decoding with the PFC scheme. Finally, we conducted a detailed experimental study on both real-world and synthetic data that reinforced some of the theoretical proofs that were derived.

The PFC scheme is able to achieve a very high degree of error stability because it uses information from every source data element (albeit in a condensed form), unlike its Wavelet counterpart. This aspect is important to Data Analysts as the answers to their queries are within a predictable and small margin from the true values.

The scheme as presented here was customized to answer multi-dimensional aggregate queries efficiently through the use of a chunking technique. The chunks stored aggregate sums which removed the need for decoding large sections of the query.

However the basic compression algorithm that we presented can be used in other contexts such as Image data compression. Image data is interesting as pixel values in close proximity tend to be highly clustered in value space and so we would expect to obtain higher data compression ratios than with numeric warehouse data. Another area for future investigation would be to test the effect of replacing the Elias coder with other variable length encoders in the post-processing phase of the PFC encoding scheme.

REFERENCES

Andrews, G. (1994). Number Theory, Dover Publications, New York.

Albrecht J., Bauer A., Deyerling O., Günzel W., Hümmer W., Lehner W., & Schlesinger L. (1999). Management of multidimensional aggregates for Efficient online analytical processing, Proceedings of the International Symposium Proceedings on Database Engineering and Applications, 1999.

Alfeld, P. University of Utah, US, online at http://www.math.utah.edu/~alfeld/math/primelist.html

U.S. Census Bureau. Census bureau databases, online at http://www.census.gov/

Chakrabarti K., & Garofalakis M. (2000). Approximate Query processing Using Wavelets, Proceedings of the 26th VLDB Conference, pp 111-122.

Chen Z., & Chen L., et al. (2003). Recent Progress on Selected Topics in Database Research, J Computer Science & Technology, 18(5), pp 538-552.

Cormack G. (1985). Data Compression on a Database System, Communications of the ACM, 28(12), pp 1336 – 1342.

Cunningham C., Song I., & Chen P. (2006) Data Warehouse Design to Support Customer Relationship Management Analyses, Journal of Database Management, Hershey, 17(2), pp 62-84.

Elias P. (1975). Universal Codeword Sets and Representations of the Integers, IEEE Transactions on Information Theory, 21(2), pp 194-203.

Elmasri R., & Navathe S. (2003). Fundamentals of Database Systems, Addison Wesley.

Garofalakis M., & Gibbons P. (2004). Probabilistic wavelet synopses, ACM Transactions on Database Systems, 29(1), pp 43-90.

Gemulla R., Lehner W., & Haas P.J., (2007), Maintaining bounded-size sample synopses of evolving datasets, VLDB Journal, Special Issue, 2007.

Hellerstein J.M., Haas P.J., & Wang H.J. (1997), Online Aggregation, Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data, pp 171-182.

Ho C., & Agrawal R. (1997). Range Queries in OLAP data cubes, Proceedings of the 1997 ACM SIGMOD Conference on Management of Data, pp 73-88.

Hunt J., Vo K.-P., & Tichy W. F., (1998). Delta algorithms: An empirical analysis. ACM Trans. Softw. Eng. Method. 7, 2, pp 192–214.

Langdon G. (1984). An Introduction to Arithmetic Coding, IBM J. Research and Development, 28(2), pp 135-149.

Lazaridis I., & Mehrotra S. (2001). Progressive Approximate Aggregate Queries with a MultiResolution Tree Structure, Proc. of SIGMOD, 2001, pp 401-412.

Lempel A., & Ziv J. (1977). A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, 23(3), pp 337-343.

Formatted: German (Germany)

Formatted: German (Germany)

Matias Y., & J Vitter J. (1998). Wavelet-Based Histograms for Selectivity Estimation, Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, June 1998, pp 448-459.
Ng W., & Ravishankar C. (1997). Block-Oriented Compression Techniques for Large Statistical Databases, IEEE Transactions on Knowledge and Data Engineering, 9(2), pp 314-328.
OLAP Report, online at http://www.olapreport.com/DatabaseExplosion.htm
Oracle 9i. (2005). Oracle Data Warehousing Guide, Oracle Corporation.
Park M., & Lee S. (2005). A Cache Optimized Multidimensional Index in Disk-Based Environments, IEICE Trans. Inf. & Syst, 88(8), 2005.
Pears R., & Houliston B. (2007). Optimization of Aggregate Queries in Data Warehouses, Journal of Database Management, IDEA Group Publishing, USA, 18(1), 2007.
Poosala V., & Gnati V. (1999). Fast Approximate Answers to Aggregate Queries on a Data Cube, Proceedings of 1999 International Conference on Scientific and Statistical Database Management, pp 24-33.
Sarawagi S., & Stonebraker M. (1994). Efficient organization of large multidimensional arrays, Proceedings of the 11th Annual IEEE Conference on Data Engineering, pp 328-336.
Shim J., Song S., Yoo J., & Min Y. (2004). An efficient cache conscious multi-dimensional index structure, Information Processing Letters, 92, pp 133-142, 2004.
Sismannis Y., Deligiannakis A. (2002). Dwarf: shrinking the PetaCube, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp 464-475.
Triantafillakis A., Kanellis P., & Martakos D.(2004) Data Warehouse Interoperability for the Extended Enterprise, Journal of Database Management, Hershey, 15(3), pp 73-84.
Vitter J., & Wang M. (1998). Data Cube Approximation and Histograms via Wavelets, Proceedings of the Seventh International Conference on Information and Knowledge Management, Washington D.C, pp 96-104.
Vitter J., & Wang M. (1999). Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets, Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, pp 193-204.
Wang, W., Feng J., Lu H., & J. X. Yu. J. (2002). Condensed Cube: An Effective Approach to Reducing Data Cube Size. In Proc. of ICDE, pages 155-165, San Jose, California, USA, 2002, pp 155-165.
Zhen H., & Darmont J. (2005). Evaluating the Dynamic Behavior of Database Applications, Journal of Database Management, Hershey, 16(2), pp 21-45.

Formatted: German (Germany)