COMPRESSING DATA CUBE IN PARALLEL OLAP SYSTEMS

Frank Dehne[1], Todd Eavis[2], and Boyong Liang[3*]

[1]School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, Canada K1S 5B6
Email: frank@dehne.net
[2]Computer Science Software Engineering, Concordia University, 1455 De Maisonneuve Blvd. West, Montreal, Canada, H3G 1M8
Email: eavis@cs.concordia.ca
*[*3]School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, Canada K1S 5B6*
Email: byliang@connect.carleton.ca

ABSTRACT

This paper proposes an efficient algorithm to compress the cubes in the progress of the parallel data cube generation. This low overhead compression mechanism provides block-by-block and record-by-record compression by using tuple difference coding techniques, thereby maximizing the compression ratio and minimizing the decompression penalty at run-time. The experimental results demonstrate that the typical compression ratio is about 30:1 without sacrificing running time. This paper also demonstrates that the compression method is suitable for Hilbert Space Filling Curve, a mechanism widely used in multi-dimensional indexing.

Keywords: OLAP, Data cube, Compressing, Parallel processing

1 INTRODUCTION

Data warehouses provide the primary support for Decision Support Systems (DSS) and Business Intelligence (BI) systems. One of the most interesting recent themes in this area has been the computation and manipulation of the data cube, a relational model that can be used to support On-Line Analytical Processing (OLAP). Data cube-based OLAP systems pre-compute multiple views of selected data by aggregating values across all possible attribute combinations (a group-by in database terminology). For a d-dimensional input set, there are 2^d possible group-bys. The resulting data structures can then be used to dramatically accelerate visualization and query tasks associated with large information sets [30].

Within the context of massive data volumes, data cube computation has to be very efficient with respect to speed and space. Many research studies have shown that parallel computation effectively speeds up data cube construction. Data cube compression, on the other hand, not only is crucial for computing and storing data cubes in limited space but also reduces I/O access time. Though compression algorithms are quite common in the literature, most are poorly suited to database/cube environments as they (i) offer relatively poor compression ratios or (ii) result in significant run-time penalties [20, 37, 39, 40].

This paper focuses on the investigation of even more efficient data cube compression techniques in the context of parallel OLAP computation systems, the Parallel Algorithms for New Data Warehousing Architectures (PANDA) [12, 13, 14, 15, 16] framework in particular. We propose an efficient data cube compression algorithm, XTDC—Extended Tuple Differential Data Cube Coding, as well as a group of corresponding data structures that can be employed in the context of high performance parallel OLAP computation. This paper also demonstrates that the XTDC method can be applied to the Hilbert Space Filling Curve - an appealing mechanism for multi-dimensional indexing frameworks [29, 35, 48]. We also propose two data cube computation algorithms, random query and sub cube construction, based on the compressed form of the XTDC data structure. The experimental results show that the typical compression ratio for a full data cube in the PANDA system, in which the fact table has 10 dimensions and 10^6 tuples, is 29.4 to 1 (without sacrificing running time). The dimensional data reduction is from 9778MB to 332MB (96.6%). The single view compression ratios are between 26 and 51 to 1.

The paper is organized as follows. Section 2 reviews the most interesting database compression techniques. Section 3 proposes the efficient data cube compression algorithm, XTDC, and corresponding data structures as well as two data cube computation algorithms based on the compressed data structure. This section also applies the Hilbert Space Filling Curve technique to XTDC. Section 4 presents the performance analysis, and Section 5 concludes the paper and discusses possible extensions of our methods.

2 RELATED WORK

In the context of data warehouse applications, we are only interested in lossless data compression techniques, which allow the original data to be fully recovered from its compressed form. There are two very important properties of conventional lossless data compression techniques: 1). Data is processed serially (FIFO), and 2). The encoder and decoder share the same data model [17, 24, 25, 44, 45, and 49]. We note that the random access of databases, such as tuple *query*, *insertion*, *deletion*, and *update,* conflicts with the serial processing and consistent data model properties of traditional data compression.

Database compression techniques have been researched since the1990's [4, 5, 9, 10, 21, 34, 37, 39, 40, 41, and 47]. Some of them are more interested in page (block) level compression. For example, Oracle applies a block-based dictionary compression technique, which reaches about a 3.1 compression ratio for a database of 55GB of data without a performance penalty in data warehouse applications [39]. Another actively researched class of database compression solutions tries to find more efficient data distributions from the characteristics and knowledge of the relation, thus achieving high compression ratios. Also the tuple-structure of a relation is preserved in its compressed form in order to support high performance database operations through the avoidance of unnecessary compression and decompression.

Ray, Haritsa, and Seshadri proposed the Column Based Attribute Compression algorithm (COLA) in 1995 [40]. COLA uses a separate frequency distribution table for each attribute in a relation. The experimental compression ratio is 21.27% for a synthetic numeric relation. Bit compression (BIT) is a well-known technique that represents each numerical attribute in bits instead of bytes. Goldstein, Ramakrishman, and Shaft proposed a derivative algorithm of BIT in 1998 by compressing relations in blocks [20] (we refer to it as Block-BIT in this paper). The typical compression ratios on real data sets are between 3 and 4 to 1. The CPU cost of decompressing a relation is approximately 1/10 the CPU cost of GZIP.

Ng and Ravishankar proposed a block-oriented database compression technique, the Tuple Differential Coding (TDC) method [37]. In TDC, all attributes in a relation are mapped into numeral domains. Tuples are converted into ordinal numbers in ascending mixed-radix order. A compressed block only stores the value of the first tuple as a reference. Each succeeding tuple is replaced by its difference with respect to its preceding tuple. The i^{th} tuple in the relation can be reconstructed from the first tuple and the first $i-1$ difference values. TDC uses the difference in the tuple level, which also keeps the tuple information of each attribute. The typical compression ratios of TDC are between 4 and 6.6 to 1 for the tables with 10^6 tuples with 8 dimensions [37].

3 DATA CUBE COMPRESSION

The multi-dimensional model is the most popular model used in data warehousing environments to support OLAP operations [30]. Data cubes, generated from fact tables, consist of the surrogate keys of the dimensional tables plus the measure fields. These surrogate keys are usually consecutive integers, which are automatically generated during the data warehouse Extract, Transform, and Load (ETL) stage. Considering the properties of the data warehouse and data cube computation, we propose the eXtended Tuple Differential Data Cube Coding (XTDC) strategies that build upon a number of ideas of existing techniques, specifically, TDC [37], BIT, and Block-BIT [20]:

1). Treat dimensional data and measure data separately. The main I/O access task is related to the large views, which have a large number of records and high dimensions. The major objective in compressing data cubes is therefore to reduce the storage of dimensional data. Given the characteristics of data warehouse applications, the dimensional data are usually represented as integers. The compression ratio mentioned later in this paper is the ratio of the size of the original dimensional data divided by the size of the compressed dimensional data.

2). Compute tuple differences to compress the dimensional data at the block level. XTDC uses the fundamental idea of the Tuple Differential Coding [37] to code the tuples in block wise fashion. Each tuple is represented using the difference between it and its preceding tuple. In order to avoid the risk of data overflow in the case of large views with high dimensions, two tuple operations, *tuple-plus* and *tuple-minus*, are proposed to support a wider range and faster data computation.

3). Compact the differences into bits.

4). Compact all the differences (the compressed dimensional data) together to remove gaps caused by *byte-alignment*. All the measure data are stored in the second part of the block.

5). Dynamically determine the number of tuples to be compressed into one block according to the value of the maximum difference in one block

6). Use a *counter mechanism* to represent consecutive *1-differences*. For those views that have low dimensions but a large number of tuples, there is very high probability that the difference values of conjunctive tuples are 1's because the attribute values of each dimension are usually consecutive integers.

7). Keep the compression information in each block. The information, such as the number of tuples in the current block and number of bits of each difference value, are stored in the block header. They are dynamically calculated during the compression process and is used during decompression.

XTDC is a block-level lossless data cube compression technique. It uses the knowledge of the characteristics of the multi-dimensional data model to guide the compression and decompression processes. XTDC preserves tuple structure in compressed views in order to get the benefits of database compression. The following section describes the details of the XTDC compacted data structure.

3.1 The XTDC Compacted Data Structure

XTDC keeps the compression information in each block, namely the *block header*, in order to preserve the tuple-structure in compressed data for high performance data access and to compress data cubes in block level. In this way, each compressed data block contains all necessary information to decompress this block or to localize the required data (tuples) from the compressed data directly. As Table 3.1 presents, a typical structure of a compressed block consists of three parts:

1). The ***Block header*** contains the compression information for this block. The length and the content of the block header may vary according to the different compression algorithms. It is also good for efficient indexing to store the uncompressed first tuple in the block header.

2). The ***Dimensional data area*** contains in bit form all the compressed dimensional data — the difference values — of the tuples in this block, in order to avoid the spare bits between tuples that may be caused by byte-alignment.

3). The ***Measure data area*** contains all the measure data in original form (uncompressed format). The offset of this segment in the current block is given by: *measure offset* = *length of block header* + ⌈(*number of tuples* − *counter*) × (*number of bits for dimensional data*)/8⌉. The offset of the measure data of ith tuple is (*measure offset*)+(*i*−*1*)×(*number of bytes for measure data*).

Table 3.1. The XTDC data structure of a compressed data view block

Block header	Length of block header
	Number of tuples for this block
	Number of bits for dimensional data of each tuple
	Number of bytes for measure data of each tuple
	Counter
	First tuple in uncompressed format
Dimensional data	Compacted tuple differences (in bit)
	…
Measure data (Uncompressed)	Measure data of 2nd tuple
	…

Because most data cube operations are read-only in data warehouse applications, XTDC focuses on storing as many encoded tuples in a block as possible, rather than designing a more flexible data structure for update operations. The following section discusses the details of using this data structure in the XTDC algorithms.

3.2 The Extended Tuple Differential Data Cube Coding Algorithm – XTDC

As we discussed previously, the principle idea of the tuple differential coding algorithms is to store the difference values of the consecutive tuples. The mixed-radix values of tuples can be calculated according to Eq.(3.1) of Definition 1[37].

Definition 1 *A relational scheme* **R**=< A_1,A_2, ...,A_n > *is a sequence of attribute domains, where* $A_i = \{0, 1, ..., |A_i| - 1\}$ *for* $1 \le i \le n$. *The value of one tuple* < a_1, a_2, ..., a_n > *is defined as*:

$$\varphi\langle a_1, a_2, ..., a_n\rangle = \sum_{i=1}^{n}\left(a_i \prod_{j=i+1}^{n} |A_j| \right) \tag{3.1}$$

In enterprise level data warehousing environments, a view to be compressed may have high dimensions with high cardinalities. Consequently, there is a very high risk of data overflow when mapping each tuple to a mixed-radix value in such environments. However, when views are fully sorted, the differences between two conjunctive tuples are usually very small. We propose two tuple operators, *tuple_plus* and *tuple_minus*, in order to encode views safely and efficiently. Theorem 1 gives the principle of the operators.

Theorem 1 *Given two consecutive tuples*: < a_1, a_2, ..., a_n > *and* < a'_1, a'_2, ..., a'_n >, *the difference value of these two tuples is*:

$$\varphi\langle a'_1, a'_2, ..., a'_n\rangle - \varphi\langle a_1, a_2, ..., a_n\rangle = \sum_{i=1}^{n}\left((a'_i - a_i) \prod_{j=i+1}^{n} |A_j| \right) \tag{3.2}$$

Algorithm 1 calculates the difference by directly manipulating the attribute values of the tuples to avoid data overflow. It is also efficient because it reduces the multiplicative operations to addition operations.

Algorithm 1. Tuple Minus

```
Input: Two tuples $T_1$, $T_2$ with d-dimension;
        Cardinalities (C[i]) for each attribute domain (A[i])
Output: The difference between the mixed-radix value of $T_2$ and $T_1$
1: difference = 0;
2: for i = 0 to d − 1 do
3:     difference = difference * C[i] + (T2[i] − T1[i]);
4: end for
5: return difference;
```

During the decompression process of XTDC, we can exploit the fact that the preceding tuple has already been decoded into its uncompressed format when decoding the current tuple. We propose the *tuple_plus* operator to operate directly on attributes of tuples in order to avoid computing the mixed-radix values.

XTDC is a block-level compression technique using the XTDC compacted data structure. The encoded dimensional data, the differences, are compacted by bits and grouped together to save maximal space. The block information is stored in the block header. The number of bits for each tuple difference value is dynamically determined by the maximum value of the differences. Algorithm 2 presents the details of the XTDC data cube compression algorithm.

Algorithm 2. XTDC Data Cube Compression Algorithm

```
Input: A view (in buf) to be compressed and its metadata
Output: The Compressed view (out buf)
1: Create a block header contains the first tuple;
```

```
2: index = 0; processed_tuples = 0; counter = 0;
3: for all tuple[i] of in_buf do
4:    difference = tuple_minus(tuple[i], tuple[i − 1]);
5:    Compute the number of tuples in this block;
6:    if tuple[i] is not the last tuple and can be fit in the block then
7:       if consecutively difference == 1 then
8:          counter + +;
9:       else
10:         difference_buf[index + +] = difference;
11:      end if
12:      max_difference = max(difference,max_difference);
13:      tuples_per_block + +;
14:   else
15:      Compute offset of measure data in this block;
16:      for j = 1 to tuples_per_block do
17:         if (j > counter) then
18:            compact different_buf[j] into $\log_2$(max_difference) bits in in_buf;
19:         end if
20:         out_buf[offset + +] = in buf[processed_tuples + j, dimension − 1];
21:      end for
22:      complete current block-header;
23:      processed_tuples+ = tuples_per_block;
24:      if tuple[i] is not the last tuple then
25:         copy tuple[i] to new block-header;
26:         index = 0; counter = 0; tuples_per_block = 1;
27:      end if
28:   end if
29: end for
```

For each block, XTDC employs two phases:

1). The **Computation phase:** XTDC calculates the differences of conjunctive tuples by using the *tuple_minus* (line 4) and dynamically computes the number of tuples in one compressed block (line 5). XTDC checks every tuple to determine if it can fit in the current block according to the changing value of the maximum difference (*max_difference*) and the number of consecutive 1's(*counter*). The differences are stored in a buffer (*difference_buf*) in integer form in this phase.

2). The **Compact phase:** After collecting enough differences for one block (or all tuples have been encoded), XTDC computes the offset (*offset*) of the measure segment of current block (line 15). The measure data is copied to the measure area of the block in uncompressed format (line 20). All of the differences, calculated in the first phase, are compacted in bit format into the dimensional area of the block (line18). Each of these differences occupies $\lceil \log_2(max\_difference) \rceil$ bits. Finally, XTDC completes the block header (line 22) and starts to compute the next block.

Note that the XTDC technique supports access to each tuple at the block level without loading the whole view and decompressing it. However, in this particular project, we use the XTDC interface to decompress the whole view immediately after loading it into main memory. So, in our project, the XTDC decompression algorithm loads the whole compressed view and retrieves the compression information from the block header. It then computes the tuples one by one using the *tuple_plus*, and simply copies the measure data to output buffer. The next block is processed (if necessary) when all tuples of the current block are decoded.

3.3 Applying the Hilbert Space Filling Curve Technique to XTDC

The XTDC algorithm uses the tuple differential coding method to compress data cubes. Tuples are sorted in a specific order and then converted into an integer representation. The difference between consecutive integers is calculated and used to represent the original data. The method that performs the integer mapping must be "one-to-one" in order to convert (decompress) the compressed integer back to a unique tuple representation. The Hilbert

Space Filling Curves technique traces a unique pathway though the points of a multidimensional space. In this sense, it may be used to provide a clean one-to-one mapping of a tuple to its ordinal or "indexed" position in the hypercubic space. We apply the Hilbert ordering to XTDC compression process in two phases: 1). Hilbert ordering maps the tuples (dimensional data) of the views to the sequence of the Hilbert Space Filling Curve and sorts the views by these sequential values. We note that a Hilbert re-sorting is required here because the core cube aggregation algorithms use a *lowX* ordering. 2). XTDC encoding uses steps similar to the standard XTDC approach except that the *tuple_minus* is the simple integer minus, and the first tuple is stored in its Hilbert sequence value in the block header. The decompression process is composed of two phases: XTDC decoding and Hilbert de-ordering.

It is very important to note that the PANDA system utilizes the Hilbert Space Filling Curve to compute the sorted views for multidimensional indexing. As a result, there is a significant potential to improve the data cube compression performance because we effectively can get the Hilbert sorted views for "free".

3.4 Compressed Data Cube Computation

Unlike conventional data compression techniques, XTDC preserves the tuple structure in compressed data, thereby allowing the OLAP computation system to manipulate the data cube in compressed format. By avoiding unnecessary decompression and compression computations, XTDC not only reduces the storage requirement and I/O bandwidth but also reduces main memory requirements.

The XTDC algorithms are able to retrieve one single tuple from a compressed view without decoding the whole block. They also improve the quality of index structures such as B-trees and R-trees by reducing the number of leaf blocks. Algorithm 3 presents the steps of localizing a specific tuple in a compressed view.

Algorithm 3. Locating One Specific Tuple in a Compressed view.

```
Input: A compressed view in XTDC data format.
       Dimensional data of the specific tuple, t
Output: Measure data of $t$ (NULL for non-existing tuple)
1: Locate the block that may contain $t$ by checking the first tuple in block headers
2: Load the entire block into main memory
3: Compute the difference ($v$) between the required tuple ($t$) and the first tuple
4: Accumulate the first $i$ different values until we reach one that is equal to OR greater than $v$
5: Return NULL if the different value is greater than $v$
6: Compute the offset of the measure data segment in the current block according to the header
   information
7: Return the i^th measure data
```

Generating sub views from a given view (parent view) is one of the primary operations of data cube computation [16, 32]. The XTDC technique allows OLAP computation systems, such as PANDA, to compute a compressed sub view from a compressed parent view directly.

Definition 2 *Given a parent view*, **R**=< $A_1, A_2, ..., A_n$ >, φ < $a_1, a_2, ..., a_n$ > *is the value of tuple* < $a_1, a_2, ..., a_n$ > . *Its k-subview is* **R**'=< $A_1, A_2, ..., A_{k-1}, A_{k+1}, ..., A_n$ >, φ' < $a_1, a_2, ..., a_{k-1}, a_{k+1}, ..., a_n$ > *is the value of tuple* < $a_1, a_2, ..., a_{k-1}, a_{k+1}, ..., a_n$ >.

Theorem 2 *Given a parent view* R, *the tuple value*, φ', *of its k-subview*, R', *is*:

$$\varphi' = \left(\varphi \, div \prod_{l=k}^{n} |A_l| \right) \times \prod_{l=k+1}^{n} |A_l| + \varphi \bmod \prod_{j=k+1}^{n} |A_j| \qquad (3.4)$$

Algorithm 4 computes a *k-subview* from a parent view using the XTDC data structure.

Algorithm 4. Construct a Compressed *k*-subview From a Compressed Parent View.

```
Input: $V_p$, a compressed parent view in XTDC data structure.
Output: $V_s$, the compressed k-subview in XTDC data structure.
1: Initialize view buffer: view_buf.
2: repeat
3:     Load one block of $V_p$.
4:     for all tuple($t_i$) of Vp do
5:         Compute the value $v_{pi}$ of tuple $t_i$.
6:         Get measure data $m_{pi}$ of $t_i$.
7:         Compute the corresponding tuple value, $v_{si}$, in $V_s$.
8:         Accumulate the measure data of $V_s$: view_buf[$v_{si}$]+= $m_{pi}$.
9:     end for
10: until all blocks of $V_p$ are processed.
11: Construct the k-subview: Compact the view_buf in XTDC format.
```

3.5 Compressing Data Cube in the PANDA System

PANDA supports high performance parallel data cube computations. Its I/O Manager is a significant feature that handles efficient I/O access during the manipulation of massive data sets. We implement the XTDC algorithm as a *Compression Interface* and plug the interface into the I/O Manager in order to compress (and decompress) data cubes. In the data-writing phase, the tuples in the view buffer are compressed by block before they are physically written to disk. In the data-loading phase, the entire compressed view is loaded from disk and decompressed in main memory (Input Buffer). The details of the system structure and the XTDC *Compression Interface* implementation are discussed in [16, 33].

In this section, we discussed the efficient data cube compression algorithm, XTDC, and its corresponding compact data structure as well as two OLAP operations – random point query and sub view generation — based on this data structure. We also demonstrated that the XTDC algorithms can utilize the Hilbert Space Filling Curve technique. Therefore, it has potential for use in OLAP systems that use the Hilbert space technique for multidimensional indexing. Finally, we introduced the strategy of applying the XTDC algorithms to a parallel OLAP computing system – PANDA. In the next section, we will demonstrate the experimental results and evaluate the performance of the XTDC algorithms.

4 EVALUATION

This section evaluates the performance of data cube compression techniques implemented in the PANDA System. The main goal of data cube compression is to reduce the space requirements of data cube computation while maintaining reasonable response time. Our tests therefore focus on two main issues: compression ratio (CR) and compression/decompression speed. In the context of data cube compression, our implementations compress the dimensional data and leave the measure data in uncompressed form. Our evaluations use the dimensional data compression ratio, CR = (*dimension size without compression*) / (*dimension size with compression*). Both compression and decompression processes are involved in data cube computation. We use wall-clock running time to evaluate the speed performance for both single view computation and full data cube computation.

In the multi-dimensional model, a data cube is organized in exactly the same format as that of a conventional relational table. In order to compare the compression efficiency between XTDC and the existing database compression techniques, we implemented the TDC [37] and the BIT database compression algorithms. It is worth noting that when we apply the TDC database compression algorithm to achieve data cube compression in PANDA, we follow the fundamental ideas of [37] except that we use our tuple computation algorithms, *tuple_minus* and *tuple_plus*, to avoid data overflow during the computation of tuple differences. As proposed by [37], TDC uses Run Length Coding (RLC) to encode the number of leading zero components in each difference. In our particular implementation, we store the differences in integer form (4-byte), which costs less than using RLC encoding. We also plug an Open Source conventional compression library, BZIP [6, 43]. All of our tests were conducted on a Linux cluster, whose primary characteristics are listed below [26]:

- Linux Kernel 2.4.18-27.7.xsmp (Redhat 7.3)
- 64-processor (dual processor nodes)
- 32-node Beowulf configuration
- Gigabit Ethernet (1000Mbps)
- Switch with a 32Gbps
- Each node has a 2 GHz Intel Xeon processor, 1.5GB RAM, and 60 GB IDE disks.

We will look at a sequence of data cube compression tests, each designed to highlight one important characteristic. We evaluate fact tables with 6 to 10 dimensions. The number of tuples in these fact tables ranges from 100K to 2M. The fact tables themselves are created with PANDA's Data Generator [16] by specifying parameters such as the number of tuples in the data set, the number of dimensions, and the cardinality in each dimension. In effect, we utilize a set of base parameters and then vary exactly one of these parameters in each of the tests. These base parameters are (with defaults listed in parenthesis): a) Fact Table Size (1M) and b) Dimension Count (10). Table 4.1 lists the cardinalities used in all of our test cases.

Table 4.1. The meta data of testing data cubes

Name of Dimension	A	B	C	D	E	F	G	H	I	J
Cardinality	6	10	50	8	25	12	3	15	8	16

4.1 Single View Compression

The efficiency of compression techniques can be clearly evaluated on single view tests. We use the Data Generator [16] to generate a group of fact tables with 500K, 1M, 2M, 5M, and 10M tuples respectively. All of these fact tables have 10 dimension attributes and 1 measure field. We arbitrarily create a group of single views corresponding to each fact table by using the Partial Data Cube generation module in PANDA. Each of these views has 7 dimensions and 1 measure field. The number of tuples and the original size of these views are listed in Table 4.2. For each single view, we apply different compression techniques, including BIT, TDC, XTDC, BZIP, and Linux GZIP. Because GZIP and BZIP are global range data compression techniques, their compression ratios are computed as total compression ratio. The BZIP libraries [43] are plugged into the same test harness as BIT, TDC, and XTDC. Both options for best compression (gzip -9) and for fast compression (gzip -1) of GZIP are used to evaluate compression ratio and running speed.

Table 4.2. The data volume of views ABCDFJG

Tuples in the Fact Table	10M	5M	2M	1M	500K
Tuples in the View	6873327	4101573	1842122	959242	489775
Uncompressed Size(MB)	210	125	56	30	15

Figure 4.1 shows that the compression ratios of BIT, TDC, BZIP, and GZIP are between 5 and 12 to 1. With the increasing size of views, the ratios of conventional compression techniques (BZIP, GZIP) slightly increase because there are better data distributions in a larger range. The number of tuples in a view does not affect the compression ratio of BIT because its compression ratio is only determined by the number of bits for every tuple, which corresponds to the cardinalities of the dimensions. The compression ratio of TDC remains stable because it always stores differences in integer form, which costs 4 bytes in our system, no matter how small the differences are. The experiments show that XTDC reaches compression ratios between 26 and 51 to 1, which are much higher than the other techniques. In XTDC, the number of bits required to store the differences in a block is determined by the maximal difference of consecutive tuples in that block. Both the bit compaction technique and the counter mechanism help XTDC to reach higher compression ratios with an increasing number of tuples in a view.

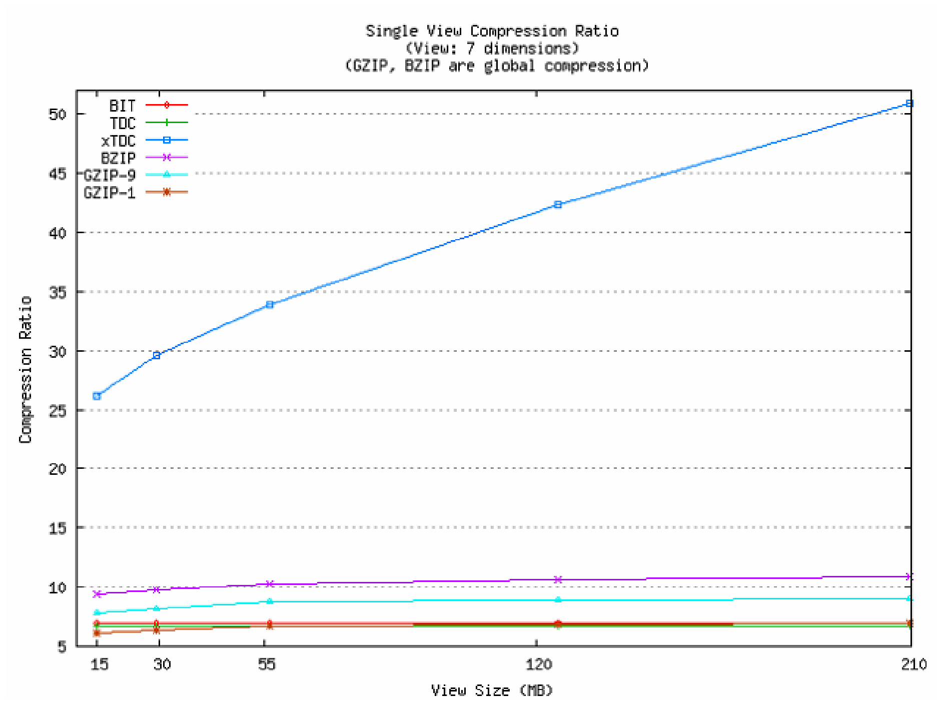

Figure 4.1. Compression ratio comparisons for single view compression

Figure 4.2 presents the total running time (compression time plus decompression time) of these compression algorithms. Data cube compression techniques (BIT, TDC, and XTDC) have the same range of running times, which are much faster than conventional ones.

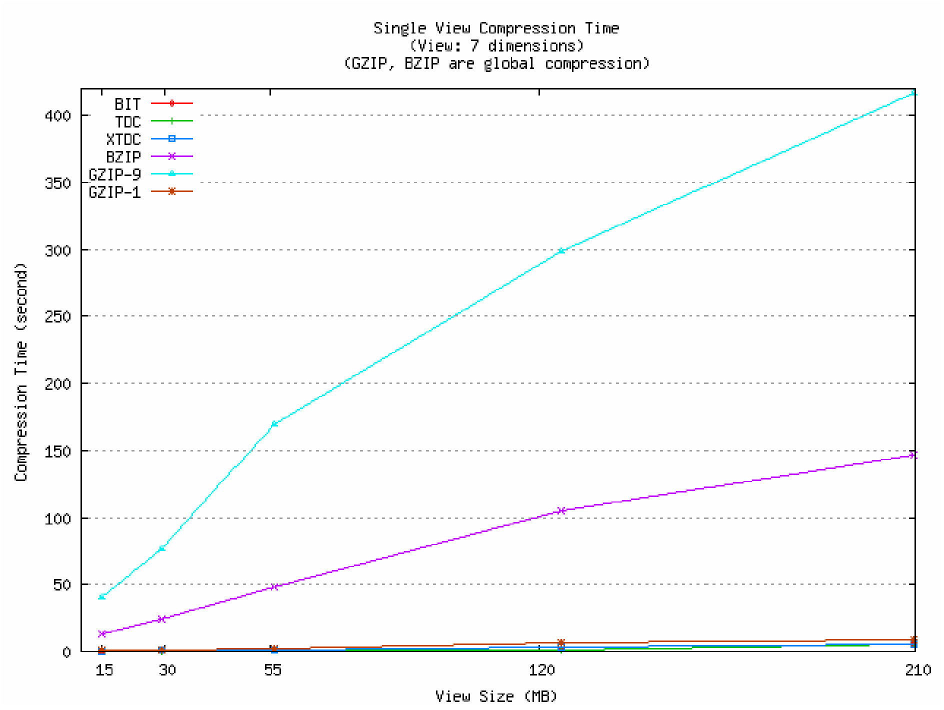

Figure 4.2. Total runtime (compression and decompression) comparison for single view compression

4.2 Full Cube Compression

In full cube tests, 2^d single views are created. These views cover all the possible combinations from 1-dimension to d-dimensions. The efficiency of single view compression will definitely affect the full cube compression. Figure 4.3 presents the average of compression ratios for full cube computation. The fact tables of these cubes have 10 dimensions and the same cardinality distribution as listed on Table 4.1. Consistent with the result in Figure 4.1, XTDC reaches a much higher compression ratio than the others (BIT, TDC) do.

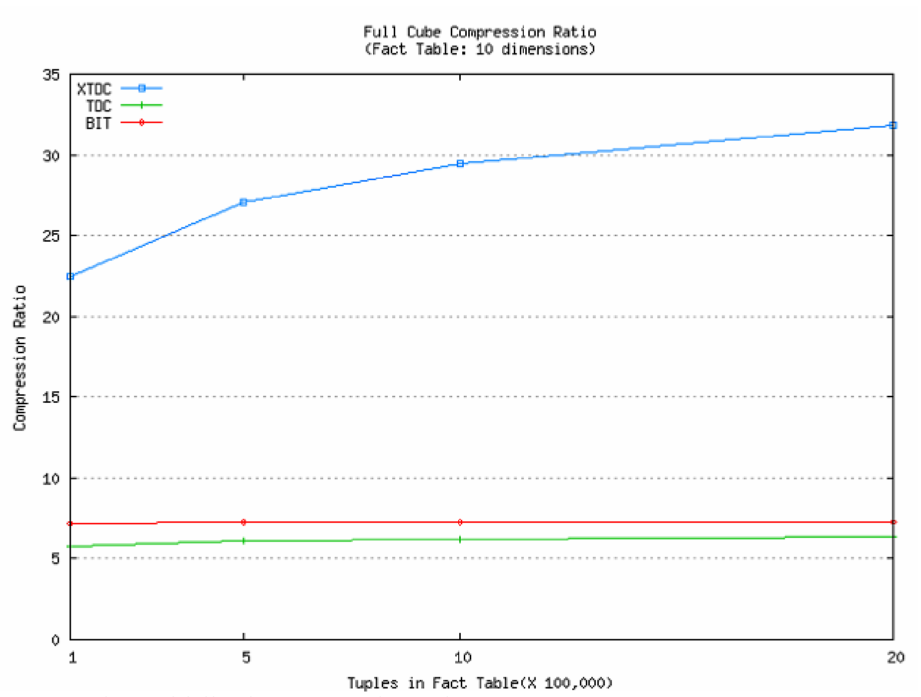

Figure 4.3. The comparison of full cube compression ratios

It is worth noting that the fully materialized data cube is much bigger than the fact table. In one of our test cases — using a fact table with 10 dimensions and 10^6 tuples — the dimensional data of the fact table is 40MB, while the total dimensional data in the full data cube generated by this fact table is 9778 MB. XTDC reaches a 29.4 to 1 compression ratio, which reduces the dimensional data from 9778 MB to 333MB. Figure 4.3 also shows that XTDC is more efficient in terms of compressing the full cube that has been generated by the fact table with the same dimensions but a larger number of tuples. In this experiment, XTDC reaches a 31.8:1 compression ratio when the fact table of the cube contains 2×10^6 tuples. Note that the compression ratios are lower on the full cube than the single views we tested in Section 4.1. As we discussed previously, XTDC uses one difference value (several bits in many cases) to represent the dimensional data of one tuple. As the number of dimensions decrease (and most views have less than the 7 dimensions used in the single view test), the ability to compress diminishes as well. Conversely, the greater the number of dimensions, the greater the benefit for compression. We also note that, as same as in single view compression, the compression ratios of BIT and TDC are not significantly affected as the number of tuples increases.

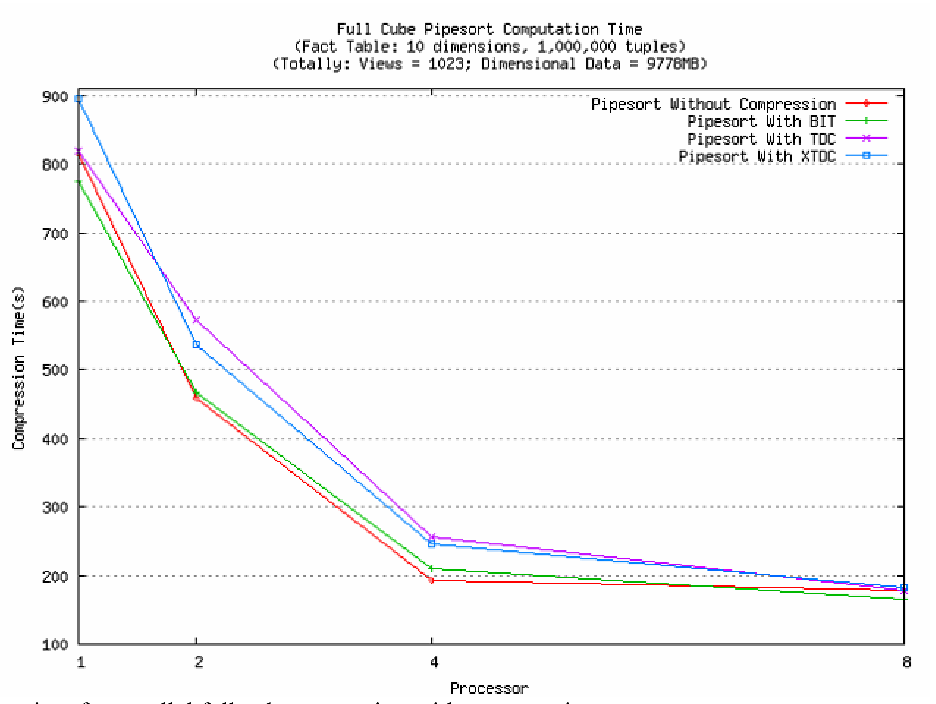

Figure 4.4. Runtime for parallel full cube generation with compression

Figure 4.4 presents the speedup of PANDA with data cube compression on multiple processors. The result shows that XTDC, as well as BIT and TDC, works very well with PANDA's parallel data cube computation. The running times are very close to the original ones. We do not include conventional compression techniques in the full cube tests because of the results of the single view experiments. In fact, the BZIP compression libraries significantly slow down full cube computation. In one of our experiments, PANDA with BZIP compression takes 872 seconds to generate a full cube using a fact table that has 10 dimensions and 5×10^4 tuples. The total compression ratio is 7.7 to 1. As we can see from Figure 4.3 and Figure 4.4, XTDC reaches a compression ratio of 22.5 to 1 for a larger data cube, which is generated by using a fact table with 10 dimensions and 10^6 tuples, and does so in less than 900 seconds.

5 CONCLUSION AND FUTURE WORK

This paper proposes an efficient data cube compression algorithm, XTDC, and its corresponding compact data structure. Building upon a number of existing compression algorithms, XTDC is effectively a combination of the following techniques:

1). Tuple differential coding: Tuples in the views are mapped to integer values and the differences of the conjunctive tuples are used to represent the views.

2). Bit compaction: The tuple differences are stored in bit form and are compacted.

3). Block-wise compression: The tuples are compressed in blocks (pages). This not only increases the compression ratio by reducing the value range of the differences but also makes more efficient data access since all the compression information is localized in individual blocks.

4). Handling dimensional data and measure data separately to remove the gaps caused by *byte-alignment*.

5). Using meta data information: Knowledge of the data cube is used when compressing and decompressing.

The XTDC technique preserves the tuple structure in compressed data cubes. Its data structure makes the compressed blocks accessible to common indexing methods such as B-trees or the packed R-trees that are actually used by PANDA. Because all information about tuples is encoded in individual blocks, the data cube operation can be done when the data cubes are still compressed. We propose two algorithms for random access and sub cube generation based on compressed data, which shows the possibility of manipulating the compressed data cube without decoding the whole views, thereby improving OLAP computing performance. We also discuss that the Hilbert Space Filling Curve technique is well suited to the XTDC algorithms. Therefore, the XTDC technique has great potential for use in practical cube systems that use space filling curves for multidimensional indexing.

The experimental results show that the XTDC technique is well suited for parallel OLAP computing systems. By integrating the XTDC algorithms into the PANDA system, the storage space requirements for OLAP computation are greatly reduced with very little performance penalty. The typical compression ratio is 29.4 to 1 for a full cube generation, in which the fact table has 10 dimensions and 10^6 tuples. The dimensional data reduction is from 9778MB to 332MB (96.6%). The experiments also demonstrate that the XTDC algorithms have the ability to achieve higher compression ratios for larger data cubes which have more dimensions and more tuples.

Because the XTDC technique preserves the structural information of the data cube in compressed form, it would be possible to extend the data cube operations on compressed data using the current design. First, it is very convenient to index compressed data blocks for fast random tuple location as the first tuple of a block is stored in the block header in uncompressed format. Because the number of blocks is reduced for the compressed view, the size of the index file is significantly decreased as well. Second, implementing compressed data cube computing inside PANDA will not only save main memory space during data cube computation but also avoid most of the data compression and decompression processes. Third, as we noted, the XTDC algorithms is suitable to the Hilbert Space Filling Curve, and the PANDA system utilizes the Hilbert ordering to compute the sorted views for multidimensional indexing. As a result, there is a significant potential to improve the data cube compression performance because we effectively can get the Hilbert sorted views for "free". Both data cube compression and multidimensional indexing can share the full benefit of Hilbert ordering.

6 REFERENCES

[1] Adamson, C. & Venerable, M (1998) *The Data Warehouse Design solutions*. John Wiley & Sons, Inc.

[2] Bernstein, P. A., Hadzilacos, V., & N. Goodman, N. (1987) Concurrency control and recovery in database systems, 1987.

[3] Brackett, M. (1996). *The Data Warehouse Challenge*. John Wiley & Sons, New York, NY, USA.

[4] Brisaboa, N., Iglesias, E., Navarro, G., & Parama, J. (2003) An efficient compression code for text databases. In *ECIR, volume 2633 of Lecture Notes in Computer Science*, pages 468–481. Springer-Verlag.

[5] Buchsbaum, A., Caldwell, D., Church, K., Fowler, G., & Muthukrishnan, S. (2000) Engineering the compression of massive tables: an experimental approach. In *Symposium on Discrete Algorithms*, pages 175–184.

[6] Burrows, M. & Wheeler, D. J. (1994) A block-sorting lossless data compression algorithm. *Digital System Research Center Research Report,* May 1994.

[7] Chen, Y., Dehne, F., Eavis, T., & Rau-Chaplin, A. (2004) Parallel ROLAP data cube construction on shared-nothing multiprocessors. In *Distributed and Parallel Databases*, volume 15, number 3, May 2004, pages 219-236..

[8] Chen, Y., Dehne, F., Eavis, T., & Rau-Chaplin, A. (2004) Building large ROLAP data cubes in parallel. In *IDEAS*, pages 367–377.

[9] Chen, Z., Gehrke, J., & Korn, F. (2001) Query optimization in compressed database systems. In *SIGMOD Conference.*

[10] Chen, Z. & Seshadri, P. (2000) An algebraic compression framework for query results. In *ICDE*, pages 177–188.

[11] Neilson Thomas Debevoise. The Data Warehouse Method. Prentic Hall,Inc., Upper Saddle River, New Jersy, USA, 1999.

[12] Dehne, F., Eavis, T., Hambrusch, S., & Rau-Chaplin, A. (2002) Parallelizing the data cube. *International Conference on Database Theory.*

[13] Dehne, F., Eavis, T., & Rau-Chaplin, A. (2001). Coarse grained parallel online analytical processing (OLAP) for data mining. *Lecture Notes in Computer Science, 2074.*

[14] Dehne, F., Eavis, T., & Rau-Chaplin, A. (2001). Computing partial data cubes for parallel data warehousing applications. *Lecture Notes in Computer Science, 2131.*

[15] Dehne, F., Eavis, T., & Rau-Chaplin, A. (2002) Parallel multi-dimensional ROLAP indexing. In *proceedings of the 3rd IEEE/ACM International Symposuim on Cluster Computing and the Grid (CCGrid2003)*, pages 86--93, Tokyo, Japan, October 2002.

[16] Eavis., T. (2004) *Parallel OLAP Computing*. PhD thesis, Dalhousie University.

[17] Gagie. T. (2003) *Dynamic length-restricted coding*. Master's thesis, University of Toronto.

[18] Giovinazzo, W. (2000) *Object-Oriented Data Warehouse Design*. Prentice Hall, Inc.,Upper Saddle River, New Jersey, USA.

[19] Goil, S. & Choudhary, A. (1999) A parallel scalable infrastructure for OLAP and data mining. In *International Database Engineering and Application Symposium,* pages 178–186.

[20] Goldstein, J., Ramakrishnan, R., & Shaft, U. (1998) Compressing relations and indexes. In *ICDE*, pages 370–379.

[21] Graefe, G. & Shapiro, L. D. (1991) Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing,* Kansas City, MO.

[22] Gray, J. & Reuter, A. (1992) Transaction processing: Concepts and techniques. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

[23] Harinarayan, V., Rajaraman, A., & Ullman, J. (1996) Implementing data cubes efficiently. In *Proceeding ACM SIGMOD Conference*, pages 205–216.

[24] Held, G & Marshall, T. (1991) *Data Compression*. John Wiley & Sons, New York, NY, USA, third edition.

[25] Hoffman, R. (1997) *Data Compression in Digital Systems*. Chapman & Hall.

[26] HPCVL. Retrieved March 8, 2007 from the World Wide Web: http://www.hpcvl.org.

[27] Huang, X., Lu, H., & Li, Z. (1997) Computing data cubes using massively parallel processors. In *Proceeding 7th Parallel Computing Workshop (PCW'97),* Canberra, Australia.

[28] Inmon, W. H. (1992) *Building the Data Warehouse*. John Wiley & Sons.

[29] Jurgens, M. (2002) Index Structures for Data Warehouses, *Lecture Notes in Computer Science, 1859*. Springer-Verlag.

[30] Kimball, R., Reeves, L., Ross, M., & Thornthwaithe, W. (1998) *The Data Warehouse Lifecycle Toolkit*. John Wiley & Sons, Inc.

[31] Kimball, R. & and Ross, M. (2002) *The Data Warehouse Toolkit*. John Wiley & Sons, Inc, second edition.

[32] Li, J., Rotem, D., & Srivastava, J. (1999). Aggregation algorithms for very large compressed data warehouses. In *Proceeding of the 25th VLDB Conference*, pages 651–662.

[33] Liang, B. (2005) *Compressing Data Cube in Parallel OLAP Systems*. Master thesis, Carleton University.

[34] Sayed, A, Hoque, L., McGregor, D, & Wilson, J. (2002) Databases compression using an offline dictionary method. In *ADVIS, volume 2457 of Lecture Notes in Computer Science*, pages 11–20. Springer-Verlag.

[35] Moon, B, Jagadish, H., Faloutsos, C., & Saltz, J. (2001). Analysis of the clustering properties of the Hilbert space-filling curve. *Knowledge and Data Engineering, 13(1):*124–141.

[36] Ng, R., Wagner, A., & Yin, Y. (2001) Iceberg-cube computation with pc cluster. In *Proceeding of 2001 ACM SIGMOD Conference on Management of Data,* pages 25–36.

[37] Ng, W. K. & Ravishankar, C. V. (1997) Block-oriented compression techniques for large statistical databases. *Knowledge and Data Engineering, 9(2*):314–328.

[38] Panda. Project website: http://panda.cgmlab.org/.

[39] Poess, M. & Potapov, D. (2003) Data compression in oracle. In Frederick H. Lochovsky, editor, *Proceedings of the 29th VLDB Conference*, Berlin, Germany.

[40] Ray, G., Haritsa, J., & Seshadri, S. (1995) Database compression: A performance enhancement tool. In *International Conference on Management of Data*.

[41] Roth, M. & Van Horn, S. (1993) Database compression. *SIGMOD Record, 22(3),* September 1993.

[42] Sarawagi, S., Agarwal, R., & Gupta, A. (1996) On the computing the data cube*. IBM Research Report*.

[43] Seward, J. bzip2 and libbzip2, Retrieved from the World Wide Web, March 8, 2007: http://sources.redhat.com/bzip2.

[44] Storer, J. (1988) *Data Compression Methods and Theory*. Computer Science Press Inc, Rockville, Md.

[45] Wayner, P. (2000) *Compression Algorithms for Real Programmers*. Morgan Kaufmann: San Diego.

[46] GNU Website. Retrieved from the World Wide Web March 8, 2007:http://www.gnu.org.

[47] Westmann, T., Kossmann, D., Helmer, S., & Moerkotte, G. (2000) The implementation and performance of compressed databases. *SIGMOD Record, 29(3):*55–67.

[48] Yu, C. (2002) High-Dimendional Indexing, *Lecture Notes in Computer Science 2341*. Springer-Verlag.

[49] Ziv, J. & Lempel, A. (1977) A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory, 23(3)*:337–343.