Compressing Data Cube in Parallel OLAP Systems

By

Boyong Liang

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

April 30, 2005

Abstract

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 respect has been the computation and manipulation of the *data cube*, a relational model that can be used to support On-Line Analytical Processing (OLAP). Within the context of massive data volumes, data cube compression is not only crucial for computing and storing data cubes in limited space, but also reduces I/O access time.

This thesis proposes an efficient data cube compression algorithm and integrates this algorithm into the PANDA system, a mature and well-studied framework for parallel OLAP computation. The experimental results demonstrate that, within the PANDA environment, the algorithm significantly reduces data cube storage space without sacrificing running time. Experimental results also demonstrate that the proposed algorithm and its supporting data structures are well suited for use with the Hilbert Space Filling Curve, a mechanism used in the PANDA system to support the generation and manipulation of multi-dimensional data cubes.

List of Acronyms

API: Application Programming Interface

BI: Business Intelligence

BIT: Bit compression/compaction

CGM: Coarse Grained Multicomputer model

COLA: COLumn-based Attribute level non-adaptive coding

CPU:Central Processing Unit

DBMS: Database management System

DMA: Direct Memory Access

DSS: decision support system

E/R modelling: entity-relationship modelling

ETL: Extract, Transform and Load

FIFO: First In First Out

GB:Giga Byte

HPCVL: High Performance Computing Virtual Lab

I/O: Input/Output

MB: Mega Byte

OLAP: On-Line Analytical Processing

OLTP: On-Line Transaction Processing

PANDA: Parallel Algorithms for New Data warehousing Architectures

RLC: Run Length Coding

TDC: Tuple Differential Coding

XTDC: Extended Tuple Differential Data Cube Coding

Acknowledgements

I would like to gratefully acknowledge the enthusiastic supervision of Dr. Dehne and Dr. Eavis. I am grateful to Dr. Corriveau for all the emotional support during my study in the school.

I am forever indebted to my family for their understanding, endless patience and encouragement when it was most required.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The use of data warehousing has become more and more popular in both Business Intelligence (BI) and Decision Support Systems (DSS). One of the most important elements of the standard data warehouse architecture is the multidimensional data model, which consists of a fact table and a group of dimension tables that are defined according to the functional requirements of the associated organization [3, 11, 18, 30, 31]. Business data organized in multidimensional structures can be represented as *data cubes*, sets of pre-computed views of selected data that are formed by aggregating values across attribute combinations (a *group-by* in database terminology). OLAP operations, such as *roll-up*, *drill-down*, *slice* and *dice*, manipulate the data cubes in order to significantly enhance the response time of data warehouse applications. To provide acceptable performance, query resolution systems must support — either directly or indirectly — all combinations of attributes associated with the given application context. Given the size of the underlying data stores, practical

data cube systems that attempt to achieve this objective must (i) provide efficient and scalable cube generation algorithms and (ii) store vast quantities of aggregated data with a minimal storage "footprint".

In terms of the first priority, parallel or multi-processor OLAP has received considerable attention in the research community over the past seven or eight years. One of the most efficient and practical systems is PANDA — Parallel Algorithms for New Data Warehousing Architectures — which has proven to be a robust framework with highly optimized algorithms and implementation. [12, 14, 13, 15, 16]. PANDA provides high performance full data cube computation (all combinations of attributes), efficient partial cube construction (the combination of selected attributes), parallel multi-dimensional indexing facilities and a parallel query engine.

With respect to data cube compression, research has been somewhat less focused. Nevertheless, the potential benefits are significant. Aside from solving scalability problems, data cube compression has at least the following advantages in the context of data warehouse applications:

1. A wider array of hardware/software architectures are available for data warehouse systems due to the reduction of storage requirements;

2. I/O traffic, which tends to be one of the most time consuming operations, is reduced;

3. Less main memory is required for compressed data cube computation. In other words, more main memory is available for other data structures and algorithm execution;

4. More pre-computed views can be stored in the same amount of storage, thus enhancing the response time of business queries.

As previously noted, however, in current practical OLAP computation systems we have seen very little research with respect to efficient data cube compression techniques. The traditional data compression techniques, including statistical data modelling and dictionary algorithms, have of course been applied in relational database systems [5, 49, 50]. But we note that these kind of compression techniques do not preserve the structure of relations in compressed form. Therefore, the whole relation has to be decompressed when randomly localizing a small piece of data (ie., a record). The cost of maintaining the consistency of the data model is also very high since any update may change the statistics of the model. The existing database compression techniques, such as BIT, Block-BIT and TDC [20, 37, 39, 40], preserve the structure of relations and either use a simple data model or block-oriented coding to maintain consistency. By doing so, these algorithms support efficient random query and update operations, with compression ratios between 3 and 7 to 1. The compression algorithms in [20, 39] are also applied to data warehousing applications and achieve compression ratios of between 3 and 4 to 1. We will discuss the details of these techniques in subsequent chapters.

This thesis focuses on the investigation of even more efficient data cube compression techniques in the context of parallel OLAP computation systems, the PANDA framework in particular. It is worth noting that sequential data compression algorithms can be applied in PANDA's parallel OLAP computation framework since each single view is processed by one processor at a time.

1.2 Thesis Contribution

The contributions of this thesis are the following:

- This thesis proposes 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. XTDC uses knowledge of the multi-dimensional data model and preserves the tuple structure in the compressed result sets. By doing this, XTDC exploits the advantages of both database compression [4, 5, 9, 10, 21, 33, 40, 42, 49] and data cube computation.

- This thesis integrates the XTDC algorithms into the PANDA system. One conventional data compression technique and two existing database compression techniques namely BIT and TDC, are also applied within the PANDA system to compress data cubes for comparison purposes. The experimental results show that the XTDC algorithms are more efficient than simply applying existing techniques. Using the XTDC technique, 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. Low cardinality views in particular produce compression ratios as high as 1353 to 1.

- This thesis applies our XTDC methods to the Hilbert Space Filling Curve. Because of the most significant property of the Hilbert ordering — points near one another in the original multidimensional space are more likely to be close to one another in the linearly ordered space — the Hilbert curve is an appealing mechanism for multi-dimensional indexing frameworks [29, 34, 53]. Moreover, the Hilbert curve provides the kind of one to one mapping that is central to the XTDC model. For this reason, the XTDC/Hilbert combination has great potential. Our experiments demonstrate that XTDC and Hilbert ordering can

in fact be effectively combined to encode data cubes. It is also important to note that PANDA applies a fast Hilbert Space-Filling Curve algorithm to partition the multidimensional index data in its high performance parallel query engine [16]. Therefore, in the PANDA system, XTDC is capable of improving the compression performance by getting the Hilbert ordering results "for free".

- This thesis proposes two data cube computation algorithms, random query and sub cube construction, based on the compressed form of the XTDC data structure. These algorithms support the manipulation of the compressed data cube without decoding the whole views, thereby improving the OLAP computing performance.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 presents some background concepts related to data warehousing, data cubes, parallel OLAP, and data compression.
- Chapter 3 reviews related work, including the PANDA system, as well as the main aspects and techniques of database compression.
- Chapter 4 analyzes data cube compression characteristics, proposes an efficient data cube compression algorithm, XTDC, and the corresponding data structure to store the compressed data cube. This chapter applies the Hilbert Space Filling Curve technique to XTDC. It also discusses two data cube computation algorithms based on the compressed data structure.

- Chapter 5 introduces the details of integrating data cube compression algorithms into the PANDA system. One generic data compression algorithm and two existing database compression algorithms are also implemented for comparison purposes.
- Chapter 6 presents and evaluates the experimental results.
- Finally, in Chapter 7, I conclude and briefly describe some possible future work.

Chapter 2

Background

2.1 Data Warehouse

Data warehousing plays a pivotal role in high performance information retrieval and decision-making support. The classic definition of the data warehouse describes the architecture as a "*subject-oriented, time-variant, and non-volatile collection of data in support of management's decision-making process*" [28]. This stable, subject-oriented representation of data ultimately supports what is commonly referred to as On-line Analytical Processing or OLAP.

Before discussing OLAP, however, it is important to distinguish OLAP from a related form of data management known as On-line Transaction Processing. OLTP is associated with a set of *transaction systems* that are used to capture and track detailed information about day-to-day business activities. Some of the most important characteristics of OLTP systems are as follows:

- OLTP databases contain detailed, transaction-oriented records driven by individual business activities.
- OLTP systems record current information.
- OLTP systems process high volume read/write operations and require sophisticated mechanisms to ensure the transactions are characterized by *atomicity*, *consistency*, *isolation*, *durability* [2, 22].

The OLTP system's normalized design ensures transaction consistency, and optimizes the storage of the information. But its focus on the individual transaction renders the OLTP system ill suited to answer questions that are focused at higher levels, such as "What were the best-selling products last year?" In order to support management's decision-making process, data warehouses are designed to have the following basic properties [1, 28, 30]:

- Data warehouses are organized around subjects, rather than atomic transactions to suit to the problems of classification and trend analysis.
- Data warehouses represent aggregated or summarized information from a variety of sources. They house data collected over very long periods in order to provide a complete history of the activity of the organization.
- Data warehouses are tuned for read-only access.

2.2 Data Cube

Operational databases are typically designed using the *Entity-Relationship* (E/R) model. This type of database is well-tuned for transaction processing which requires

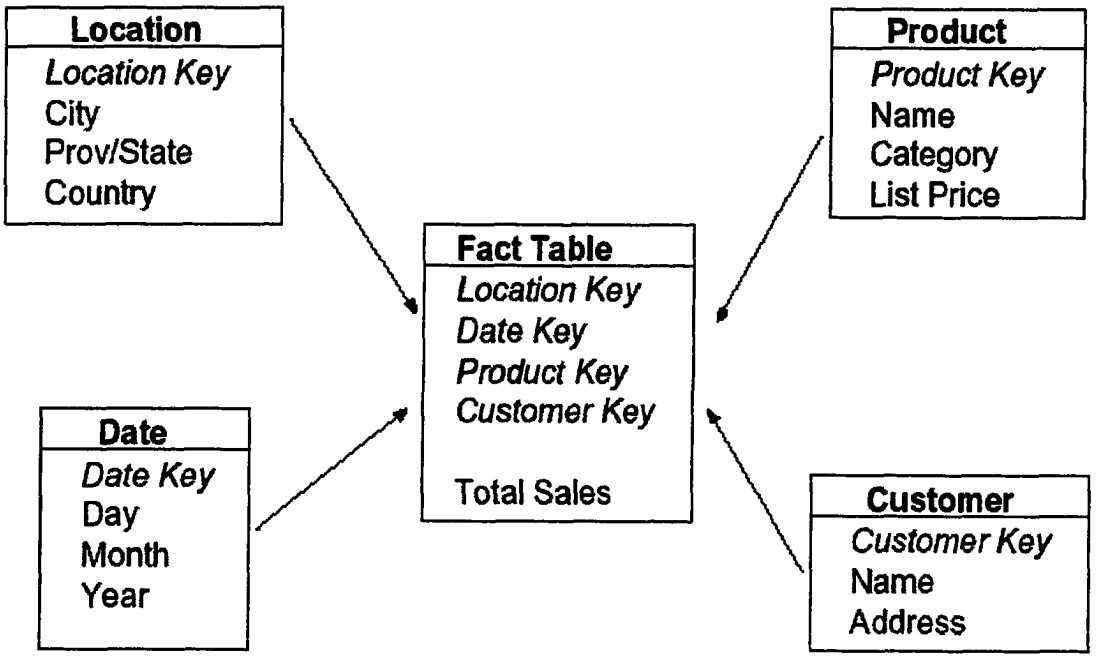

Figure 2.1: A four-dimensional Star Schema

rapid record update. However, it is not particulary well-suited for resolving complex queries in large data warehouses. One of the most successful and popular data organization frameworks for data warehouses is the *dimensional model.* The dimensional model consists of a single *fact table* with multipart keys and a set of associated *dimension tables.* Collectively, the tables form what is known as a Star Schema. Each dimension table contains a single *primary key* that corresponds exactly to one of the components of the *multipart key* in the fact table. In addition to the primary keys from the dimension tables, a fact table houses the value of one or more numeric aggregations. It is also important to note that the primary keys in dimension tables are usually meaningless surrogate keys, which are typically represented by unique, application generated integers [30]. Figure 2.1 provides a simple illustration of a Star Schema with four dimensions.

Business data organized in multi-dimensional structures can be logically visualized

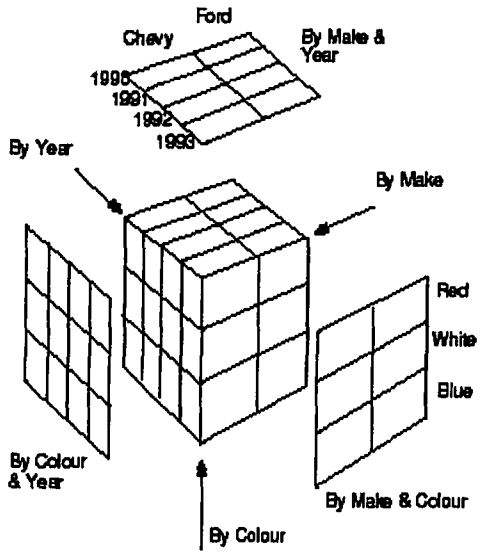

Figure 2.2: An example three dimensional data cube for automobile sales data

as data cubes, where the cells of the cube contain measure values and the edges of the cube define the natural dimensions of the data [30]. 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 $\mathbf{R}$, 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]. Figure 2.2 shows an example of a three dimensional data cube for automobile sales data.

While the data cube provides a logical representation of multi-dimensional data, it is important to emphasize that it can also be a physical representation. Specifically, in order to provide acceptable response time on cube-oriented queries, OLAP systems generally attempt to physically materialized as many of these 2^d group-bys as time and storage resources permit. In practice, however, a fully materialized data cube can be hundreds or even thousands of times larger than the original fact table

[16, 41]. For many organizations, the required resources would be prohibitively expensive. Instead, we would like to ensure that the materialized cube structures are not significantly larger than the base data, a goal that requires *selective materialization* and/or compression techniques.

2.3 OLAP computation

OLAP extends the basic reporting capabilities of traditional information processing systems to support a robust multidimensional analysis of the archived data from a variety of perspectives and hierarchies. OLAP operations manipulate the data organized in the data cube format to support analysis of the data from different perspectives. There are five fundamental OLAP query forms:

- **Roll-up**.The roll-up operation collapses the hierarchy along a particular dimension(s) so as to present the remaining dimensions at a coarser level of aggregation. Figure 2.3 illustrates how the "location" dimension, originally listed in a city-by-city fashion, is aggregated in order to provide provincial totals.

- **Drill-down**. The drill-down function allows users to obtain a more detailed view of a given dimension. Figure 2.3 shows how the "product" dimension is broken down from its initial, broad categories into product- specific listings.

- **Slice**. By extracting a slice of the original cube corresponding to a single value of a given dimension, *slice* operation allows the user to focus on values of interest. Figure 2.4 illustrates the process for a single value of the "product" dimension.

- **Dice**. The dice operation generates a *subcube* of the original space. By specifying value ranges on one or more dimensions, the user can highlight meaningful

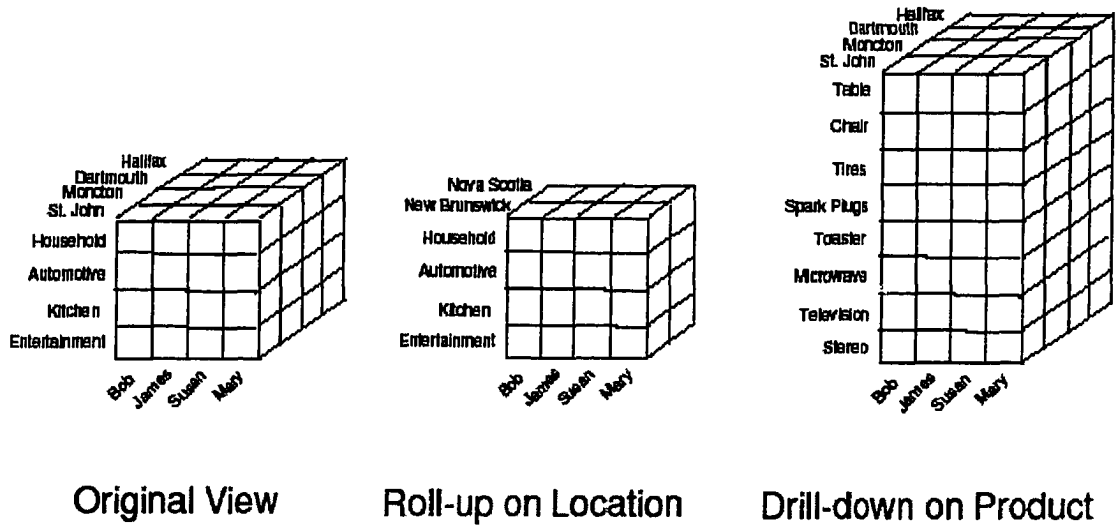

Figure 2.3: Roll-up and Drill-down on a simple three-dimensional cube

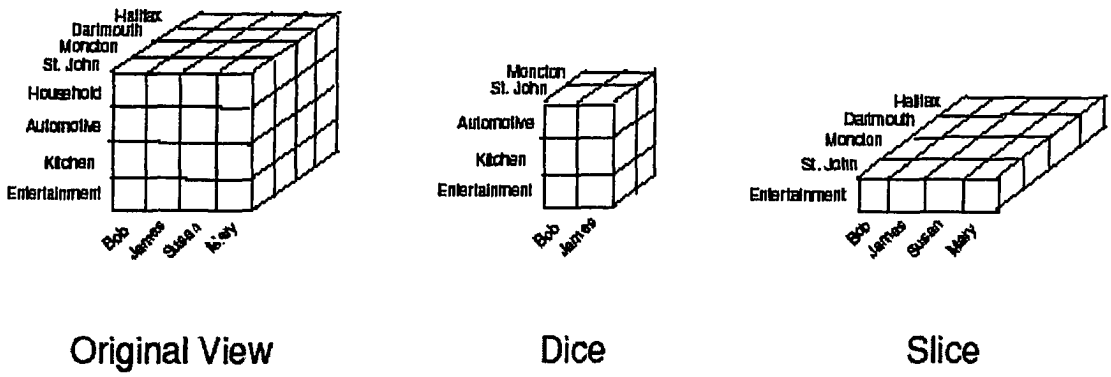

Figure 2.4: Slicing and Dicing a three-dimensional cube

blocks of aggregated data. In Figure 2.4, a subset of dimension values on "product", "location", and "customer" have produced the $3 \times 2 \times 2$ subcube.

- **Pivot.** The pivot is a simple operation that allows OLAP users to visualize cube values in more natural or intuitive ways. Figure 2.5 provides a simple example with a symmetrical $4 \times 4 \times 4$ cube.

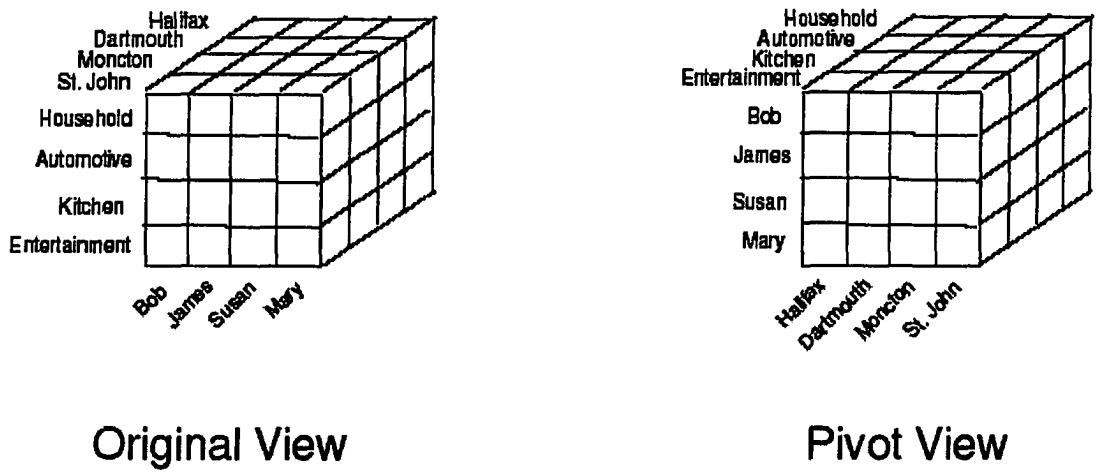

Figure 2.5: The Pivot operation

2.4 Data Compression

Data compression techniques have been widely used to reduce data storage space as well as I/O access bandwidth. In the context of data warehouse applications, it is important to note that we are only interested in lossless data compression techniques, which allow the original data to be fully recovered form its compressed form.

There is a significant variety of data compression algorithms. Current techniques of lossless data compression may be categorized into two broad classes: statistical techniques and dictionary techniques [17, 24, 25, 46, 47, 52].

- Statistical Data Modelling

  Statistical techniques separate the work of compression into two parts: statistical data modelling and coding [37]. These algorithms use variable-length bit patterns to encode individual symbols based on their frequency of occurrence.

  In 1843, S.F.B.Morse developed an efficient code to allow the electrical transmission of messages. Morse code assigns short codes to frequently transmitted letters, and longer codes to infrequently occurring letters [25], a fundamental

statistical strategy.

Huffman coding is a second well-known compression technique. Huffman code is constructed by building a binary tree where the leaves of the tree represent the probabilities of the symbols to be coded. The desirable prefix property, which assures that a codeword is not a codeword prefix for another symbol, allows the effective decoding of Huffman codes [25]. The Shannon-Fano compression algorithm is similar to Huffman coding except that it builds the code tree for each symbol "top-down" instead of "bottom-up" as the Huffman tree construction algorithm does [46].

Another widely used compression technique is *arithmetic coding*, in which the individual symbols are coded as a part of a fractional number according to their frequency of occurrence. Arithmetic coding merges entire sequences of symbols and encodes them as a single number [24, 46, 47].

- Dictionary Algorithms

  Dictionary algorithms, formalized by the work of Lempel and Ziv [54] in the 1970s, encode sequence of symbols that are found in a 'dictionary' into shorter codes. In the Lempel-Ziv algorithm, an adaptive dictionary is generated while compressing; any repeated string of symbols is replaced by a token pointing to an earlier occurrence of that string. The encoder and decoder maintain identical copies of the dictionary.

Run-length coding (RLC) is a variable-to-fixed-length, static encoding technique. A sequence of consecutive identical symbols is replaced with three elements: 1) a single symbol, 2) a run-length count, and 3) an indicator that signifies how the symbol and count are to be interpreted. It is highly effective when the data has many runs of consecutive symbols [24, 46, 47].

Most commercial compression and open source tools, such as GZIP [48], are based on the Lempel-Ziv algorithm. Statistical modelling techniques may produce superior compression, but are slower. BZIP [6, 45] is another open source compression library which uses a block-sorting algorithm to achieve speed comparable to algorithms based on the techniques of Lempel-Ziv. In addition, it obtains compression close to the best statistical modelling techniques.

There are two very important properties of these conventional data compression techniques.

1. Data is processed serially (FIFO) in either on-line or off-line compression/ decompression environments.

2. Data models, either statistical or dictionary models, must be consistent. The encoder and decoder share the same model.

2.5 Database Compression

To support high performance databases running on compressed data, the database compression techniques must allow standard database operations such as tuple *query*, *insertion*, *deletion*, and *update*. We note that the random, localized access of databases conflicts with the *serial processing* property of traditional data compression. The cost is too high to decompress a large portion of the data if only a small part of the data, i.e. one tuple, is required. It is also impractical to maintain the statistical data model or dictionary used by traditional data compression algorithm, since the entire data set has to be re-compressed during tuple update in order to preserve consistency.

In Section 3.2, we will discuss the most recent results about database compression.

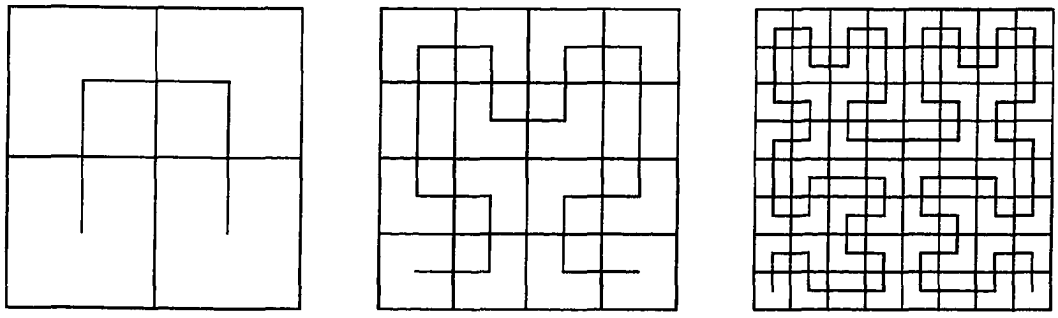

Figure 2.6: The first three steps of Hilbert space-filling curve in a two dimensional space

2.6 Hilbert Curve

Practical database systems must pay great attention to issues such as indexing and record layout. OLAP/cube systems are no exception. In fact, indexing and storage formats tend to be even more problematic due to the inherently multi-dimensional nature of user queries. To improve query response time on these multi-dimensional requests, data is often sorted or ordered in what is expected to be an advantageous manner. For example, *lowX* order has been used for this purpose [43]. lowX works as follows: for the attribute set $A_1, A_2, A_3, ..., A_d$, attribute A_1 becomes the *primary index*, A_2 becomes the *secondary index*, and so on. However, while lowX is likely to work well for those queries defined on the first dimension, it is not likely to be effective when the queries are issued on another dimension or multiple dimensions [15, 16].

In 1891, D. Hilbert introduced a continuous one dimensional curve that passes through every point of a multidimensional space [34]. One significant property of the Hilbert Space-Filling Curve is that points near one another in the original space are more likely to be closed to one another in the linearly ordered space. Figure 2.6 shows

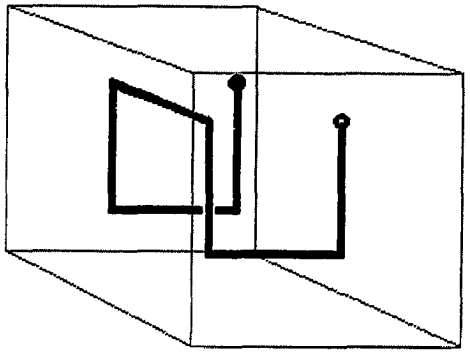

Figure 2.7: The first step of Hilbert space-filling curve in a three dimensional space

the first three steps and demonstrates how the Hilbert Curve travels every point in a two dimensional space. Figure 2.7 and Figure 2.8 present the first two steps of constructing the Hilbert Curve in three dimensional space.

Because of its ability to localize hypercubic queries in multi-dimensional space, the Hilbert curve is an appealing mechanism for multi-dimensional indexing frameworks. It was for this reason that it was selected for inclusion in the PANDA project. Consequently, an important requirement of our own compression research is the ability of the compression scheme to support data that is ordered in Hilbert order in addition to the simpler lowX format. In Section 4.4.3, we will return to this issue.

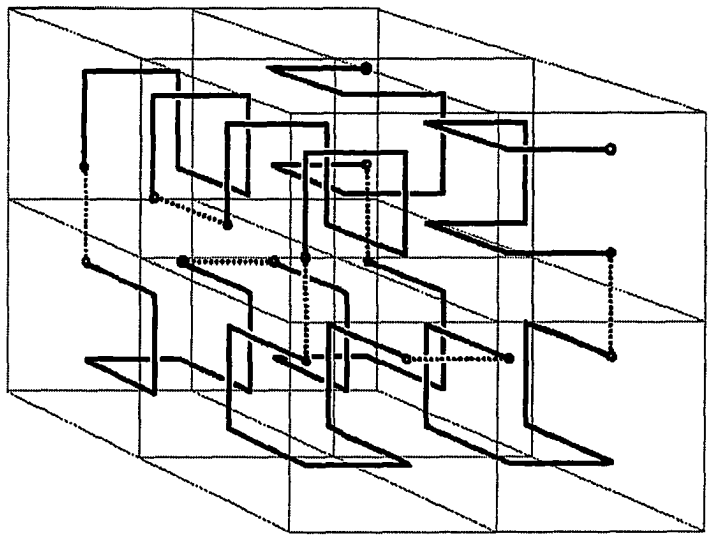

Figure 2.8: The second steps of Hilbert space-filling curve in a three dimensional space

Chapter 3

Related Work

3.1 PANDA, A Parallel OLAP Computing System

As data warehouse volumes have grown in recent years, so too has the need for scalable, highly optimized algorithms for management of that data. To this end, a small group of parallel OLAP systems or frameworks have been presented in the literature [16, 19, 27, 36]. One of the most heavily studied is the PANDA Project [12, 16, 7, 8]. It combines efficient data cube algorithms, data structures, and heavily optimized implementations into a robust parallel platform that is both load balanced and communication efficient. Key features include:

1. High performance parallel data cube generation.
   The parent/child relationships of the 2^d group-bys are often illustrated by way of the data cube lattice [23]. A simple four dimensional lattice is depicted in Figure 3.1. During the generation phase, PANDA divides the view graph into "p" equally weighted sub trees in advance, thereby allowing subsequent view construction to be fully localized. To support graph decomposition, the system

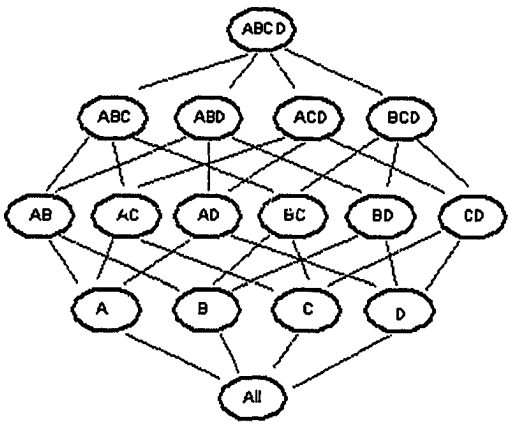

Figure 3.1: A simple four dimensional lattice

uses both a sophisticated cost model and *k-min-max* partitioning algorithm [16]. Once the scheduling information has been delivered to each node, a sequential PipeSort algorithm is used to generate the local partial cube [44]. The experimental results confirm parallel efficiency of 80% to 95% on processors counts from 1 to 24, and a near linear performance curve for increases in view count and data set size [16]. A one Terabyte data cube can be produced in less than an hour [38].

2. Efficient partial cubes parallel computing.

   PANDA proposed a suite of algorithms for partial cube construction. Based upon a greedy model, the algorithms introduce intermediate views to generate efficient 'partial cube schedule trees' in order to identify an efficient model for the generation of the selected set. Experimental evaluation has shown that the new algorithms generate trees that are 25% to 70% cheaper than those produced by more naive approaches [16].

3. Parallel multi-dimensional indexing and parallel query engine.

A parallel query engine has been designed to support two of the most important forms of OLAP queries - point query and range query. A new parallel indexing model has been proposed and implemented in order to effectively support these queries in a high dimensional OLAP context. This parallel indexing model applies a fast Hilbert Space-Filling Curve algorithm [35] to partition the multidimensional index data. Experimental evaluation demonstrates that for arbitrary queries on a 16-processor parallel machine, the corresponding speedup for parallel query resolution is close to linear. In a high volume environment, the parallel query engine can resolve 1000 random multi-dimension range queries on a 20GB data cube in less than 10 seconds [16].

4. Support for contemporary parallel computing platforms.
   PANDA algorithm design is based on the *Coarse Grained Multicomputer* (CGM) model of parallel computing [13, 16], a model intended to capture the characteristics of current, practical computing architectures.

   An I/O Manager was designed to control I/O operations in the PANDA system for efficient high volume I/O access. By managing a group of view buffers, the I/O Manager tells the operating system when, where, and how to physically write aggregated data to disk. The benefits of using the I/O Manager are as follows:

   (a) Minimizing the I/O bottleneck by carefully regulating the flow of data through the OS. In addition, by writing buffers in large chunks, view blocks are written contiguously, thereby improving performance for record retrieval.

   (b) Using separate I/O threads and computation threads to enjoy the benefits of the disk controller's high performance features, Direct Memory Access (DMA) in particular.

(c) Making PANDA flexible enough to allow new I/O facilities to be plugged in. Data cube compression algorithms are an obvious example.

PANDA provides a data generator that produces a wide variety of integer-based data sets that are representative of common data warehouse applications. The generator accepts user-defined parameters, such as the number of records in the data set and the number of unique values (cardinalities) in each dimension. One of the dimensions is treated as the measure field.

During the past four years, the PANDA Project has grown from a single algorithm to a suite of algorithms that collectively supports our data cube parallelization efforts. It is an ideal experimental platform for implementing data cube compression.

3.2 Database Compression

This section describes some of the most important existing techniques for database compression. As described in the previous chapter, we are only interested in lossless data compression techniques in data warehouse applications.

A variety of database compression techniques have been researched since the 1990's. Many papers have recommended using database compression to achieve high database operation performance [49, 50].

One class of solutions focuses on the application of traditional data compression techniques in relational database systems at four different levels, namely the file (relation) level, the page (block) level, the record (tuple) level, and attribute level. Statistical data modelling and dictionary backward referencing tend to achieve higher compression ratios in a larger data range, but more substantial costs for re-compression and re-decompression have to be paid to achieve random localization

and data model consistency. Many methods of this form attempt to strike a balance between high compression ratios and efficient database operations.

Some of them are more interested in page level compression. The experimental results show that it is advantageous to spend some compression and decompression time to get more data in the same physical disk I/O access [49]. Since a block is a unit of I/O access, there is no extra I/O cost for maintaining the data model's consistency.

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]. Instead of having a single frequency distribution for the entire relation, COLA uses a separate frequency distribution table for each attribute in a relation, since data belonging to the same attribute is usually semantically related and is expected to have a similar distribution. The experimental compression percentage is 21.27% for a synthetic numeric relation. The size of each tuple is 76 bytes and the relation has 9964 records.

Bit compression (BIT) is a well-known technique that represents each numerical attribute in bits, instead of bytes. If a numerical attribute domain has size k containing values 0 to $k-1$, it needs $\lceil \log_2(k) \rceil$ bits to represent each attribute value, yielding a saving of $sizeof(int) - \lceil \log_2(k) \rceil$. Both compression and decompression

computations with the BIT technique are very straightforward without needing to maintain a complicated data model.

The BIT technique maintains relation structure information in the smallest unit, the attribute level. Let **R** be a relation with d attribute fields $\{A_1, A_2, ..., A_d\}$, to store one attribute A_i requires $B_i = \lceil \log_2(\max(A_i)) \rceil$ bits, where i is from 1 to d, $\max(A_i)$ is the maximum attribute value of i^{th} field in **R**. Coding each attribute value (A_i) in a fixed number of bits (B_i), the number of bits for each tuple $(\sum_{i=1}^{d} B_i)$ is fixed as well.

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 thesis). The idea of Block-BIT is as follows. Each compressed block stores a 'frame of reference', the minima and maxima of all dimensions in a relation. All attributes for each tuple in the compressed block are presented by storing just enough bits to distinguish between the values in this range. This method supports individual tuple, and even individual field (add the difference to the minimum reference of this field) decompression. The typical compression ratios on real data sets are between 3 and 4 to 1. It reaches as high as 88 to 1 on low cardinality data sets. The CPU cost of decompressing a relation is approximately 1/10 the CPU cost of GZIP.

Block-BIT reduces the value of B_i by partitioning tuples in blocks. For each attribute, only the difference between the current attribute value and the smallest value in the same field inside the current block is coded. By doing so, it reduces the B_i to $\log_2(\max(A_i) - \min(A_i))$. Where $\max(A_i)$ and $\min(A_i)$ are the maximum and minimum attribute values of i^{th} field in the range of the current block.

3.2.1 Tuple Differential Coding

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. Algorithm 1 describes the main steps of the TDC compression techniques. 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].

Instead of coding tuples by using differences for each attribute as Block-BIT does, TDC uses the difference in the tuple level, which also keeps the tuple information of each attribute. In fact, the tuple level difference value removes the duplicated attribute values, and only keeps the different attribute values and compacts them into one integer and, as such, compresses the whole relation.

Wu and Ravishankar propose a theoretical basis for the performance characteristics of TDC [51]. They provide methods for estimating the compression for cases where the population from which database records are sampled is either uniform, Zipf, or the product of a uniform distribution and an arbitrary distribution.

In the context of data warehouse compression, we are more interested in database compression techniques that focus on numeric domains, since the dimensional attributes in data cubes are integers. To pursue high database operation performance, it is crucial to preserve the table structures in compressed data in order to randomly localize and to access data without decompressing a large amount of data.

By exploiting these database compression techniques in the PANDA system, we

Algorithm 1 TDC Compression Algorithm: Ng and Ravishankar

Input: A relation with d numeric attribute fields

Output: A compressed relation

1: Attribute domain ranking: re-rank the lexicographical orderings of the attributes to pursue smaller difference among the tuples.
2: Tuple re-ordering: convert each tuple to a unique integer. Re-order the tuples in ascending order.
3: Block partition: Partition the reordered table into disjoint blocks of tuples depending the size of a memory page.
4: Block encoding: Using the first tuple as a reference, each succeeding tuple is replaced by its difference with respect to its preceding tuple. Using RLC encode the number of leading zero components in each difference.

have been able to focus our own attention more completely on data cube compression characteristics and data cube compression algorithms. The following chapters discuss data cube compression methods and the implementation of these techniques within the PANDA system to achieve high efficiency data cube computation.

Chapter 4

Data Cube Compression

Compressing data in a data warehouse system is attractive for two reasons: data storage reduction and performance improvement. Storage reduction is a direct and obvious benefit, while performance improvement is obtained for the following reasons. First, a reduction in physical I/O potentially improves the performance of operations on the data warehouse because smaller units of data need to be loaded into main memory. Second, because of the reduction of loaded data, the compressed data cube computation requires fewer main memory resources. The computation can be accelerated because of more flexible main memory usage patterns and the avoidance of memory swapping. Third, it is also important to note that more views of the data cube can be pre-computed and stored because of the compression, thus improving data warehouse performance.

This chapter proposes an efficient data cube compression algorithm - XTDC. The major contributions of the current thesis can be highlighted as follows:

1. Discusses the characteristics of data cube computation and compression.

2. Proposes an efficient data cube compression algorithm, XTDC, and the corresponding data structure. The method builds upon existing data compression and database compression techniques.

3. Introduces tuple difference operations to avoid computation overflow.

4. Applies the Hilbert Space Filling Curve technique to XTDC.

5. Introduces a number of basic compressed data cube operations based on the XTDC data structure.

4.1 Data Cube Compression Characteristics

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 can identify the following key characteristics of the data cube compression process.

1. Data cube computation involves high volumes of data,

2. Meta data, such as the number of dimensions and the cardinality for each dimension, is known in advance and managed by the data cube computation systems.

3. Each view of a data cube is a conventional relation (table). It consists of dimensional fields (attributes), which are integers, and one or more than one measure fields, which can be integers, floats, or other data types.

4. Data in a data cube is very stable. In practice, it is updated in *batch* at fixed but fairly infrequent intervals (e.g., weekly).

5. Data warehouse systems support random localized data accesses, such as queries, based on data cubes. The compressed views of a data cube should preserve the relation structure information in order to achieve high performance.

6. The original data should be retrieved from the compressed data cube without losing information.

7. Indexing compressed data is required.

According to these characteristics and the existing database compression techniques, we propose an efficient data cube compression algorithm, which reaches higher compression ratios than the existing data compression and database compression techniques do. The record identity is also preserved within the compressed data, so that the compressed data can be directly utilized in data cube computation.

4.2 The XTDC Data Cube Compression Strategy

We propose an efficient data cube compression algorithm, eXtended Tuple Differential Data Cube Coding (XTDC), that builds upon a number of existing techniques. Specifically, XTDC is based on a number of the fundamental ideas of TDC [37], BIT and Block-BIT [20]. A special data structure is also proposed in order to support high compression ratios. The basic compression strategy of XTDC can be described as follows:

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. In these large views, the dimensional data makes up the bulk of the every record. The major objective in compressing data cubes is therefore to reduce the storage of dimensional data. Given the characteristics of data warehouse applications (i.e., dimensions represent categorical variables), the dimensional data are usually represented as integers. On the other hand, the type of the measure data might vary, and include integer, float and double data types. The compression models of mix-typed data we have seen so far are complicated and the compression ratios are not very high, (a special compression technique can represent eight byte floating point numbers by using four bytes) [49]. In order to pursue higher compression ratios and high computing performance, we focus on compressing dimensional data and leave the measure data in the original (uncompressed) format. The compression ratio mentioned later in this thesis 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 basic idea of the Tuple Differential Coding method [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-add* and *tuple-minus*, are proposed to support a wider range and faster data computation. These differences of conjunctive tuples are stored in a buffer and are compacted into bits during the next step.

3. Compact the differences into bits.

   In the context of data cube construction, the views (sub cubes) are sorted by the dimensional data fields. The dimensional data, which are surrogate keys,

are consecutive integers that are generated in the ETL stage. The differences between conjunctive tuples are usually very small. XTDC stores these differences in bits instead of bytes. The number of bits for each tuple is dynamically determined by the value of the maximum difference in the current block.

4. Compact all the differences together to remove gaps caused by *byte-alignment.* In XTDC, we implement a new data structure that compacts data to allow for greater compression of the views. In this data structure, the differences are stored together in order to remove the gaps that are caused by byte-alignment. All the measure data are stored in the second part of the block. Details of the operations based on the data structure are discussed in the next section.

5. Dynamically determine the number of tuples to be compressed into one block. In XTDC, the number of tuples that can be compressed in one block cannot be predicted, since the number of bits for each tuple is determined by the value of the maximum difference in this block. In order to encode as many tuples as possible into one block, the number of tuples in one block is dynamically computed during the compression process.

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 and the views are sorted by their attribute fields. XTDC uses a counter to represent these consecutive 1's in each data block in order to improve compression efficiency in this case.

7. Keep the compression information in each block. The data compression 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. This information is dynamically calculated during the compression process and is used during decompression. The details are discussed in the next section.

In data cube computation systems, the cube *meta data*, such as number of dimensions and cardinalities of each dimension, is maintained by the computation system and is shared amongst all the views of the data cube. XTDC gets this meta information from the system instead of storing it in each compressed view. All information required to reconstruct the original views is therefore either stored inside the blocks or is available in the data cube computation system.

XTDC is a block-level lossless data cube compression technique. It takes advantage of the knowledge of the characteristics of the multi-dimensional data model to pursue high compression efficiency. XTDC uses this meta data to guide the compression and decompression processes, rather than analyzing the data distribution statistics as with most conventional data compression techniques. XTDC preserves tuple structure in compressed views in order to get the benefits of database compression, such as random data access, and to support high performance data cube computation. The details of the XTDC algorithms and their implementation will be discussed in Section 4.4. The following section describes the details of the XTDC compacted data structure.

4.3 The XTDC Compacted Data Structure

In order to preserve the tuple-structure in compressed data, the database compression techniques such as BIT, Block-BIT [20] and TDC [37], compress relations by putting

all attribute values of one tuple together. The precondition is that all fields of the relation are, or can be matched to, integers. As we discussed in characteristic 3 of Section 4.1, data types of measure data of data cubes may vary. It is impractical to map these "continuous" data types, namely float and double, to integers for the purpose of data compression. The COLA technique [40], which treats each attribute domain separately in order to find more efficient distribution, gives us a hint to compress all the attribute fields and store the measure data separately in order to pursue high compression ratios.

Based upon the core requirements presented in the preceding section, we propose a new data structure corresponding to our data cube compression algorithm. According to characteristic 5 in Section 4.1, XTDC preserves the tuple-structure of compressed relations for high performance data access. Since it is a block wise compression technique, XTDC keeps the compression information in each block, namely the *block header*. 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. It is also good for efficient indexing to store the uncompressed first tuple in the block header. As described previously, XTDC compresses dimensional data separately and leaves the measure data in uncompressed form (these measure data can be compressed later if necessary). Each tuple is encoded by its difference value and stored in bit form. In order to avoid the spare bits between tuples that may be caused by byte-alignment issues, this data structure stores all of the compressed dimensional data together and puts all of the measure data in the remaining space in the same block. By doing this, XTDC not only reaches a higher compression ratio by removing the gaps, but also preserves the tuple-structure at the block level.

Table 4.1 presents the typical structure of a compressed block. It consists of three parts:

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 2^{nd} tuple
	...

Table 4.1: The XTDC data structure of a compressed data view block

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.

2. The **Dimensional data area** contains in bit form all the compressed dimensional data — the difference values — of the tuples in this block.

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: $measure\ offset = length\ of\ block\ header + \lceil (number\ of\ tuples - counter) \times (number\ of\ bits\ for\ dimensional\ data)/8 \rceil$. The offset of the measure data of i^{th} tuple is $(measure\ offset) + (i-1) \times (number\ of\ bytes\ for\ measure\ data)$.

As we described in Section 4.1, 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, namely tuple delete and insertion. (As noted above, tuple updates are normally done in batch mode.) The following sections will explain the details of this data structure

when implementing the XTDC algorithms in the PANDA system. Here, we briefly demonstrate how this data structure works using an example.

Example 4.3.1 presents two different solutions for compressing a block of tuples using TDC techniques. In solution 1, the dimensional data and measure data of each tuple are combined. There is a 7-bit gap caused by byte-alignment in each compressed tuple. Solution 2 demonstrates how the byte-alignment gaps are removed by using this data structure. As a result, more tuples can be compressed in a block of the same size.

Example 4.3.1 *Remove gaps caused by byte-alignment*
Suppose after computing a group of tuples, the maximum tuple difference is 300, which means the dimensional data of each tuple can be stored in 9 bits ($\lceil \log_2(300) \rceil$). If the measure data of each tuple holds 4 bytes and the block size is 4KB, in which 32 bytes are used as the block header (including the first tuple of this block), the following two approaches are possible:

Solution 1: Put dimensional data and measure data together.
Since memory on modern architectures is accessed on byte boundaries, we require 2 bytes to store one difference value. Each tuple therefore needs $2+4=6$ bytes due to the byte-alignment issue. 7 bits are wasted in each tuple. The number of tuples that can be compressed in this block is $(4096-32)/6+1=678$.
Solution 2: Put dimensional data and measure data into two separate segments.
Let $x+1$ be the number of tuples that can be compressed in this block. The size of the dimensional data segment in this block is $\lceil (x \times 9)/8 \rceil$. The size of the measure data segment is $x \times 4$. We have $4096 >= 32+(x \times 9)/8+x \times 4$ The number of tuples can be compressed in this block is as large as 793.

Since each difference occupies a fixed number of bits, manipulating the data within

the current block is all that is necessary to retrieve the uncompressed tuple. Due to the variety of data types the measure fields may have, it is impractical to manipulate these measure fields by bit to remove the gaps as with solution 1. Another reason we prefer the second solution is that, by putting measure data together, there is a better chance to compress these fields later because of a similar distribution model in a relatively large data range [40]. Furthermore, it is better to choose those compression techniques that can guarantee the compression ratio in advance, such as the floating point numbers compression algorithm [49].

4.4 The Extended Tuple Differential Data Cube Coding Algorithm - XTDC

This section discusses the details of the XTDC data cube compression algorithms.

4.4.1 Tuple operators

As we discussed previously, the principle idea of the tuple differential coding algorithms is to store the difference values of the consecutive tuples. In [37], the TDC algorithm calculates the mixed-radix values of tuples according to Equation 4.1 of Definition 1

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

$$\varphi < a_1, a_2, ..., a_n >= \sum_{i=1}^{n} \left(a_i \prod_{j=i+1}^{n} |A_j| \right) \tag{4.1}$$

In enterprise level data warehousing environments, a view to be compressed may have high dimensions with high cardinalities. The potential range of the mixed-radix values of tuples is equivalent to the product of the attribute 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_add* and *tuple_minus*, in order to encode views safely and efficiently. Theorem 1 presents 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:*

$$\varphi < a'_1, a'_2, ..., a'_n > -\varphi < a_1, a_2, ..., a_n >= \sum_{i=1}^{n} \left[(a'_i - a_i) \prod_{j=i+1}^{n} |A_j| \right] \tag{4.2}$$

Proof. Apply Equation 4.1 of Definition 1 to the left hand side of Equation 4.2, we have:

$$\varphi < a'_1, a'_2, ..., a'_n > -\varphi < a_1, a_2, ..., a_n >= \sum_{i=1}^{n} \left(a'_i \prod_{j=i+1}^{n} |A_j| \right) - \sum_{i=1}^{n} \left(a_i \prod_{j=i+1}^{n} |A_j| \right) \tag{4.3}$$

Combining the right hand side of Equation 4.3, we get the right hand side of Equation 4.2.

□

Algorithm 2 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. Considering the millions of calls to this operation in large view compression, the performance improvement is significant.

Algorithm 2 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. Note that even when disk based indexing is used, records within a given block are un-indexed and must be assessed sequentially. Algorithm 3 directly operates on attributes of tuples to avoid computing the mixed-radix values.

It is worth noting that a high performance decompression algorithm is also very important to the current project. As we will discuss in the next chapter, the XTDC compression and decompression interface is plugged directly into PANDA. The cubes are compressed before being written to disk and are decompressed immediately after being loaded to memory. It is also important to note that the XTDC technique provides the potential to compute child views in compressed form without decompressing the designated parent view. We will discuss this potential benefit in detail in Section 4.5.

4.4.2 The XTDC Algorithms

XTDC is a block-level compression technique using the XTDC compacted data structure we proposed in Section 4.2. The encoded dimensional data, the differences, are

Algorithm 3 Tuple Add

Input: A tuple T_1 with d-dimension and an integer of $difference$;
Cardinalities ($C[i]$) for each attribute domain ($A[i]$)

Output: A tuple T_2 whose value is greater in $difference$ than T_1's

```
1: fact = difference; inc = 0;
2: for i = d − 1 downto 0 do
3:    if fact ≠ 0 then
4:       reminder = fact%C[i];
5:       fact/= C[i];
6:    else
7:       reminder = 0;
8:    end if
9:    T2[i] = T1[i] + reminder + inc;
10:   if T2[i] > C[i] then
11:      T2[i]− = C[i];
12:      inc = 1;
13:   else
14:      inc = 0;
15:   end if
16: end for
```

compacted by bit and grouped together to save maximal space. The block information, such as the number of tuples per block and the first tuple (in uncompressed format) of the current block are stored in the block header. The number of bits for each tuple difference value is dynamically determined by the maximum value of the differences, $\lceil \log_2 max\_difference \rceil$. Therefore, given the fixed block size, the number of tuples per block varies according to the number of bits per tuple during compression. A counter of consecutive single increments of tuples is stored in the block header as well. In some cases, such as compressing views that have a large number of tuples but relatively low dimensions and cardinalities, it is most likely that the values of consecutive tuples are consecutive integers, which means the differences are 1's. Encoding these kinds of tuples by using counters can significantly increase the compression ratios in these cases. Example 4.4.1 presents the details steps of using XTDC. The contents of a compressed block are listed in Figure 4.4.

As Algorithm 4 presents, the XTDC data cube compression algorithm dynamically determines how many tuples can be compressed in one block. For each block, XTDC employs two phases:

1. The **Computation phase** collects all the information for one block. XTDC calculates the differences of conjunctive tuples by using the *tuple_minus* operation shown in Algorithm 2 (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 of those tuples are stored in a buffer($difference\_buf$) in integer form in the first phase.

2. The **Compact phase** creates a compressed block. After collecting enough differences values for one block (or all tuples have been encoded), XTDC computes

the offset of the measure segment of current block (line 15). This offset($offset$) is calculated according to the block header information as described in Section 4.3. 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 (line 18). 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.

Algorithm 5 shows the XTDC decompression process. 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, the algorithm presented here loads the whole compressed view. In any case, XTDC retrieves the compression information from the block header (line 3), and computes the $offset$ of the measure segment of the current block (line 4, same as Algorithm 4). The first tuple is retrieved from the block header as well (line 5) and each tuple is computed by adding the difference value to its preceding tuple using the *tuple_add* (line 7,12) shown in Algorithm 3. The differences are either 1's (for the first *counter* tuples) or else are retrieved from the dimensional area (line 11). The measure data is simply copied to the output buffer (line 8, 13). The next block is processed (if necessary) when all tuples of the current block are decoded.

Example 4.4.1 demonstrates the details of compressing a single view by using XTDC.

Example 4.4.1 *The XTDC data cube compression processes.*
Suppose the block size is 40 Bytes, the size of the 'block header' is 32 Bytes, each

Algorithm 4 XTDC Data Cube Compression Algorithm

Input: A view (in_buf)to be compressed and its meta data
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 log2(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
```

Algorithm 5 XTDC Data Cube Decompression Algorithm

Input: A compressed view(in_buf) and its meta data
Output: The decompressed view (out_buf)

1: $processed\_tuples = 0$;
2: **while** $processed\_tuples < number\_of\_tuples$ **do**
3: Get block header information;
4: Compute $offset$ of the measure data;
5: $out\_buf[processed\_tuples++] = get\_first\_tuple\_from\_header()$;
6: **for** $i = 1$ to $counter$ **do**
7: $out\_buf[processed\_tuples + i] = tuple\_add(out\_buf[processed\_tuples + i - 1], 1)$;
8: $out\_buf[processed\_tuples + i, dimension - 1] = in\_buf[offset++]$
9: **end for**
10: **for** $i = counter$ to $tuples\_per\_block$ **do**
11: $difference = convert\_to\_integer(in\_buf[(i-counter)*bits\_of\_difference])$;
12: $out\_buf[processed\_tuples + i] = tuple\_add(out\_buf[processed\_tuples + i - 1], difference)$
13: $output[processed\_tuples + i, dimension - 1] = in\_buf[offset++]$;
14: **end for**
15: $processed\_tuples += tuples\_per\_block$
16: $in\_buf += offset$;
17: **end while**

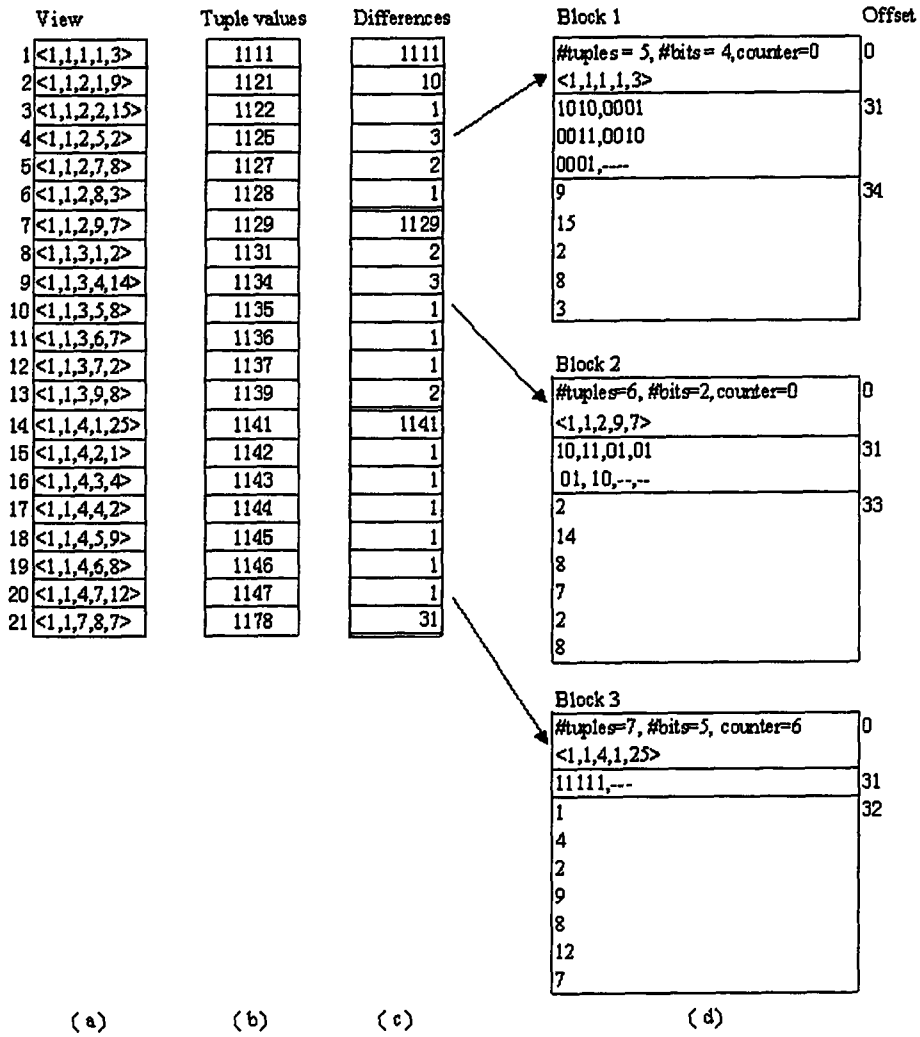

Figure 4.1: An example of the XTDC compression process

measure data fits in 1 Byte, and the cardinality of every dimension is 10. A view (sorted by dimensional data) shown in Figure 4.1(a) is compressed as follows:

- *The tuple values, shown in Figure 4.1(b), are calculated during the compression process.*
- *The number of tuples that can fit in one block are dynamically computed during compression. In this case, since the length of the block header is 32 Bytes for all the blocks, the dimensional data area and measure data area are 8 bytes in total. For the first 6 tuples, the maximum difference is 10, as Figure 4.1(c) shows. Therefore, the number of bits required for each difference is 4. As a result, tuple-1 to tuple-6 are compressed in Block-1. Specifically, tuple-1 is stored in block header, the difference values of tuple-2 to tuple-6 are compacted in 3 bytes, and the measure data are copied to the measure data area that contains 5 bytes, as shown in Figure 4.1(d)*
- *When there are consecutive 1-differences that start from the second tuple of the current block (like tuple-15 to tuple-20), XTDC uses a 'counter' to represent those tuples (as Block-3 shows)*

4.4.3 Applying the Hilbert Space Filling Curve Technique to the XTDC Algorithms

The XTDC algorithm uses the tuple differential coding method to compress data cubes. Tuples are sorted in a specific order, 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"; otherwise, it would not be possible to convert the compressed integer back to

a unique tuple representation. Similar to the mixed-radix order method, which has been used in the XTDC algorithms, 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.

As previously noted, the PANDA system successfully utilizes Hilbert curves within its indexing framework [15, 16]. In this section, we look at how XTDC can be efficiently integrated with the PANDA's Hilbert curve approach. To begin, we note that its compression process includes the following 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 since the core cube aggregation algorithms use a lowX ordering.

2. XTDC encoding. Uses steps similar to those found in the standard XTDC approach shown in Algorithm 4 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 phase:

1. XTDC decoding. Uses steps similar to those found in the standard XTDC approach shown in Algorithm 5 except that the *tuple_add* is the simple integer add.

2. Hilbert de-ordering. Maps the sequence of the Hilbert Space Filling Curve to the dimensional data of the tuples.

This thesis uses a fast Hilbert curve generation algorithm provided by Doug Moore [35] to support our XTDC design. Experimental results are evaluated in Chapter 6. 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 since we effectively can get the Hilbert sorted views for "free".

4.4.4 Memory Overhead

Both compression and decompression are performed at the block-level. In the XTDC algorithms, one extra memory buffer is used in both the compression and decompression processes. In the compression phase, a memory buffer, equivalent in size to one disk block, is allocated for storing the tuple differences in integer form (as line 10 shows in Algorithm 4). In the decompression phase, data is decompressed block by block. A buffer, again equivalent in size to a disk block, is allocated for storing the loaded data.

In order to make the compressed data available at the block level, XTDC puts the compression information in the block header. Typically, the space overhead of a block header is $4 \times sizeof(int)$ bytes plus the size of one tuple (for storing the first tuple) in each block. The block size may slightly affect the compression ratio for the following two reasons: (i) The smaller the block size is, the smaller the maximum difference value may be, and thus, the fewer bits that each compressed tuple needs, and the higher the compression ratio will be. (ii) The space overhead of the block header may reduce the compression ratio if the block size is small, since more blocks may be needed. Our implementation allows users to select the block size, ranging in multiples of 4KB, in order to reach high performance in various system environments.

The memory overhead of the XTDC algorithms during the compression and decompression processes is not heavy in the context of massive data cube computation. The benefits of saving main memory in the future — during data query resolution — could be significant because the data to be processed is highly compressed. Chapter 6 evaluates both the compression ratios and the response time of the XTDC algorithms.

4.5 Compressed Data Cube Computation

Unlike conventional data compression techniques, XTDC preserves the tuple structure in compressed data, thereby allowing the OLAP computation system to often 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. This section describes how the XTDC data structure supports high performance data cube computations based on compressed data. Two of the most important operations — random query and sub view generation — are discussed.

Section 4.5.1 presents an algorithm for the point query operation to show that the XTDC algorithms have the potential to query data in its compressed form. We note that most of the major OLAP operations — slice and dice, roll-up and drill down, etc — are based upon range queries. In our future research, we expect to investigate the extension of this technique to that environment. In addition, Section 4.5.2 discusses an algorithm for sub view generation, which is also based upon the manipulation of views in their compressed form.

4.5.1 Random Query

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. Because the XTDC data structure stores the first tuple of each block in the block header, thereby allowing the original values of all tuples to be determined, the index of a view can be built in the same way as in the E/R model.

Algorithm 6 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 in to main memory
3: Compute the difference (v) between the required tuple (t) and the first tuple
4: Accumulate the first i different values until it equals 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

Algorithm 6 presents the major steps of localizing a specific tuple in a compressed view. The details may vary slightly due to the different algorithms used in the actual implementation. In any case, the block header stores the information about how this block was compressed. The block which may contain the specific tuple (t) can itself be located as per ordinary indexing methods. Since the compressed view is sorted, the tuple t (if it exists) must be in the block which has the biggest header tuple value that is smaller than or equal to the value of tuple t. By using indexes, we only load the block when absolutely necessary. The difference between tuple t and the first tuple is computed using the *Tuple_Minus* method shown in algorithm 2 (line 3). The sequence number (i) of the tuple(t) is calculated by accumulating the difference values

retrieved from the dimensional data area of the block (line 4). If the summation of the difference values is greater than v, we know that the tuple t is not in this view at all. A NULL value is returned at line 5. The offset of the measure data area can be computed (line 6) according to the header information (as Section 4.3 discussed). The i^{th} value of the measure data area is the measure data of t.

Example 4.5.1 demonstrates how we can resolve a multi-dimensional point query without requiring decompression of all records.

Example 4.5.1 *Localize the tuple* $< 1, 1, 3, 4 >$ *in the view that compressed in Example 4.4.1 shown in Figure 4.1(d).*

- *Locate the block by checking the first tuple in the block header. Block 2 is loaded.*
- *Compute the difference between this tuple and the first tuple of the loaded block:* $tuple\_minus(< 1, 1, 3, 4 >, < 1, 1, 2, 9 >) = 5$.
- *Accumulate the difference value, we get the result 5 after adding the second difference.*
- *The measure data of* $< 1, 1, 3, 4 >$ *is the second entry, which is 14, in the measure data area.*

4.5.2 Sub View Generation

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. By doing so, we avoid decompressing the whole parent view and then compressing the target sub view since the output of the computation is in

compressed format. Again, the main memory requirements are reduced significantly because the compressed data is much smaller than the original one.

In the XTDC technique, we only use dimensional attributes to compute the tuple value. The tuple value of a sub view can be calculated by using the value of the corresponding tuple of its parent view.

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

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

$$\varphi' = \left(\varphi \; div \prod_{l=k}^{n} |A_l|\right) \times \prod_{l=k+1}^{n} |A_l| + \varphi \; mod \prod_{j=k+1}^{n} |A_j| \tag{4.4}$$

Proof. According to Equation 4.1 of Definition 1, the tuple value, φ', of k-subview is

$$\varphi' = \left[\sum_{i=1}^{k-1}\left(a_i \prod_{j=i+1}^{k-1} |A_j|\right)\right] \times \prod_{l=k+1}^{n} |A_l| + \sum_{i=k+1}^{n}\left(a_i \prod_{j=k+2}^{n} |A_j|\right) \tag{4.5}$$

The value, φ, of the corresponding tuple in the parent view can be expanded as:

$$\varphi = \left[\sum_{i=1}^{k-1}\left(a_i \prod_{j=i+1}^{k-1} |A_j|\right)\right] \times \prod_{l=k}^{n} |A_l| + a_k \prod_{l=k+1}^{n} |A_l| + \sum_{i=k+1}^{n}\left(a_i \prod_{j=k+2}^{n} |A_j|\right) \tag{4.6}$$

Applying Equation 4.6, we have 4.7 and 4.8:

$$\varphi \; div \prod_{l=k}^{n} |A_l| = \sum_{i=1}^{k-1}\left(a_i \prod_{j=i+1}^{k-1} |A_j|\right) \tag{4.7}$$

$$\varphi \; mod \prod_{j=k+1}^{n} |A_j| = \sum_{i=k+1}^{n}\left(a_i \prod_{j=k+2}^{n} |A_j|\right) \tag{4.8}$$

Again, applying 4.7 and 4.8 to the right hand side of 4.4, we get the right hand side of 4.5, which is φ'.

□

Algorithm 7 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 V_p **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.

Theorem 2 presents a way of computing the tuple values of a sub view without decoding the corresponding tuples of the parent view. Algorithm 7 shows the main steps to compute a *k-subview* from a parent view using the XTDC data structure. A temporary buffer(*view_buf*) is used to construct the k-subview(V_s). For each tuple (t_i) of the parent view, its value, v_{pi}, is calculated by using *Tuple_Add* in line 5. Line 7 computes the corresponding value (v_{si}) of the sub view according to Theorem 2. Line 6 locates its measure value in the measure data area as shown in algorithm 5. The measure data is summarized in line 8. After processing all of the tuples of the parent view, line 11 generates the sub view by compacting *view_buf*: For those entries with values, their indices are the values of tuples in the k-subview and their values are the measure data of the corresponding tuples. The compaction process simply calculates the differences of conjunctive indices and stores the data in XTDC format.

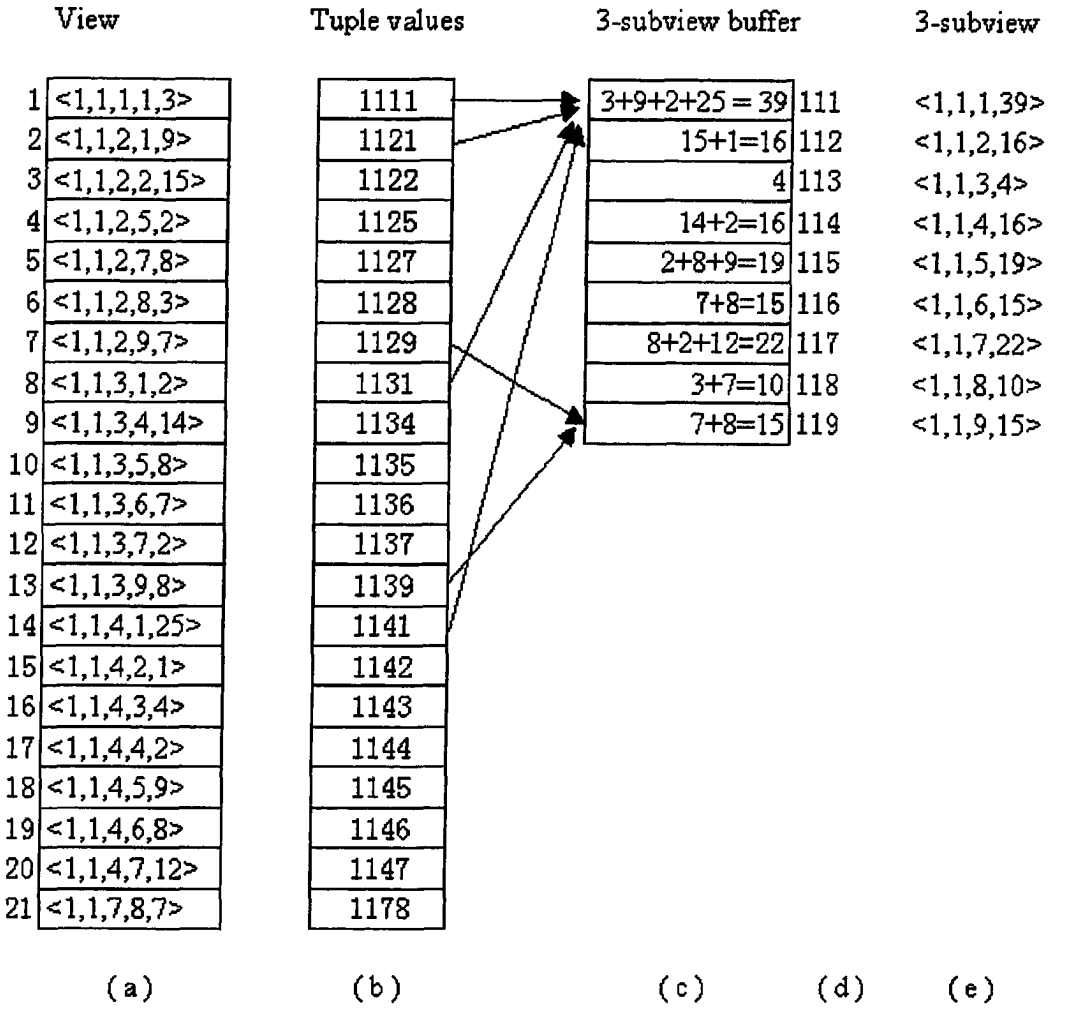

Figure 4.2: An example constructing a subview from a compressed view

Figure 4.2 shows an example of computing the 3-subview from the view that was compressed in Example 4.4.1. The indices of the 3-subview buffer (shown in Figure 4.2(c) and (d)) are the tuple values of the subview. The values of the buffer are the measure data that was summarized from the parent view. After the 3-subview buffer is completed, the subview is constructed (Figure 4.2(e)) and can then be compacted using XTDC. It is important to note that the view buffer is constructed according to Theorem 2, without decompressing the parent view.

4.6 Conclusion

In this chapter, we discussed the characteristics of data cube computation and compression. Based on these characteristics, as well as existing compression techniques, we proposed an efficient data cube compression algorithm — XTDC. Its corresponding data structure allows XTDC to retrieve every tuple from the view's compressed form at block level without decompressing the whole view. The attribute data and measure data are compressed (stored) separately in each block in order to improve the storage efficiency. We also introduced two tuple difference operations to reduce the risk of computation overflow and to improve computation performance. In addition, this chapter demonstrated that the XTDC algorithms can utilize the Hilbert Space Filling Curve technique. Therefor, it has potential for use in OLAP systems that use the Hilbert space technique for multidimensional indexing. Finally, two OLAP operations – random point query and sub view generation — based on the XTDC compressed data structure, are introduced to demonstrate XTDC's potential for the execution of efficient OLAP operations in compressed data form.

Chapter 5

Compressing Data Cube in the PANDA System

PANDA supports high performance parallel data cube computations. One of our primary objectives is to design an efficient data cube compression algorithm and to integrate it into the PANDA system. By doing so, we reduce disk space requirements during large data cube computation with no or little performance penalty, thereby making PANDA a more efficient, practical system both in terms of performance and storage efficiency.

As a robust, practical system, PANDA is supported by many available facilities. The I/O Manager is a significant feature that handles efficient I/O access during the manipulation of massive data sets. The Data Generator generates synthetic numeric relations with various user-defined parameters, such as number of dimensions, cardinalities and number of records. These features make PANDA an ideal experimental system for data cube compression research.

In this chapter, we will discuss the implementation and integration of a number of

data compression techniques — including the XTDC algorithms, the TDC and BIT database compression algorithms, and the BZIP data compression algorithm — into the PANDA system. Section 5.1 illustrates the PANDA's I/O Manager module and the strategy used to plug a *Compression Interface* into the I/O Manager in order to compress (and decompress) data cubes. By plugging in the Compression Interface, the XTDC algorithms discussed in the previous chapter can easily be incorporated into PANDA. In order to evaluate the performance of XTDC and compare its compression ratios to some of the more successful existing database compression techniques, Section 5.2 presents the details of implementing the TDC and the BIT database algorithms. The implementation of an Open Source data compression algorithm — BZIP — is also discussed in Section 5.3

5.1 Integrating Data Cube Compression Into PANDA

In the PANDA system, the parallel Pipesort data cube computation algorithm logically accesses data through view buffers. The I/O Manager manages these view buffers and handles physical disk access.

As Figure 5.1 presents, the I/O access pattern for view generation in PANDA includes the following steps [16]:

1. PANDA selects a parent view from which to compute the target view;
2. I/O Manager loads the entire parent view from disk to main memory;
3. PANDA computes the target view from the loaded data;

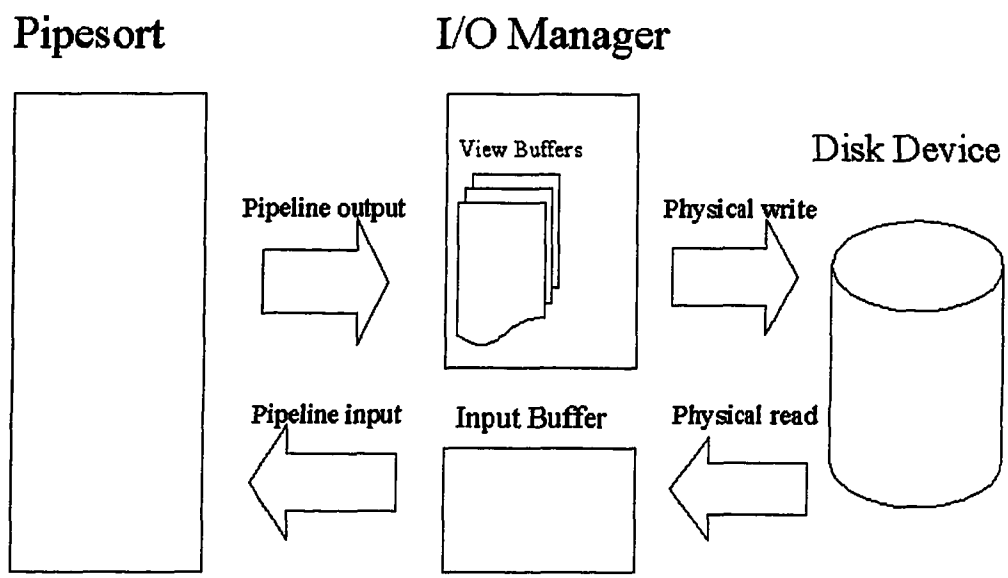

Figure 5.1: An illustration of the data cube I/O access in PANDA

4. The result of the computation (target view) is written to a view buffer tuple by tuple;

5. When the view buffer is full or the view computation finishes, the I/O Manager physically writes the data from the view buffer to disk.

Our data cube compression and decompression processes are plugged into the data-writing and data-loading interface of the I/O Manager respectively. In the data-loading phase (step 2), the entire compressed view is loaded from disk and decompressed in main memory (Input Buffer). In the data-writing phase (step 5), the tuples in the view buffer are compressed by block before they are physically written to disk. Figure 5.2 illustrates the PANDA's I/O access mechanism with respect to the data cube compression feature.

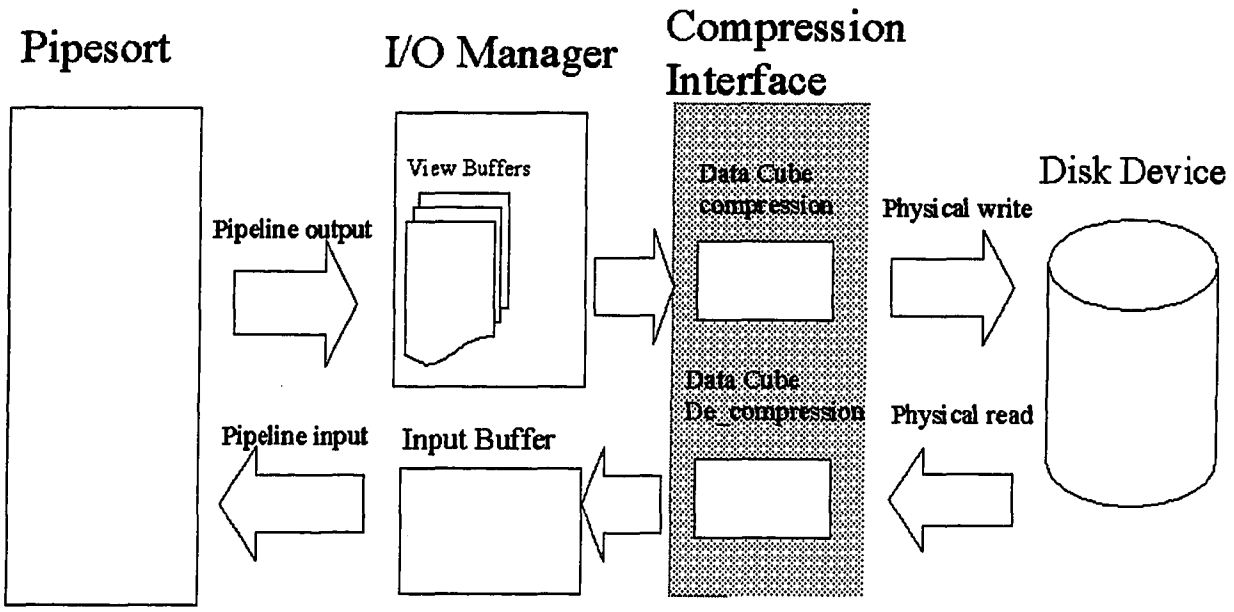

Figure 5.2: An illustration of data cube I/O with the compression interface

5.2 Applying Database Compression Techniques to Data Cube Compression

In the multi-dimensional model, a data cube is organized in exactly the same format as that of a conventional relational table. The database compression techniques can be simply applied directly to data cube compression in order to reduce the storage space. The following two sections describe the integration of the BIT and TDC [37] database compression algorithms with the PANDA system. As we discussed in previous sections, we only compress the dimensional data in the current scheme.

5.2.1 BIT Data Cube compression

Because it can be used in traditional database systems, the BIT technique can also be used to compress data cube dimensional data. The following two kinds of Meta

data, which are necessary to compress a view, are managed by the data warehouse system: (i) number of dimensions and (ii) cardinality for each dimension.

Although the cardinality values that BIT uses to compress dimensional data are at the relation level, in the BIT algorithm implementation, we compress data at the block level because of the following two reasons:

1. To follow the I/O Manager's behavior. PANDA uses the I/O Manager to physically write a view buffer to disk. The I/O Manager transfers data to disk in multiple writes if the size of the view is bigger than the view buffer.

2. To prepare the compressed views for high performance random data query, since the unit of I/O access is a block.

The BIT data cube compression algorithm is presented in Algorithm 8. The algorithm works by compacting dimension attributes ($tuple[i, j]$) into a fixed number of bits ($bits\_of\_dimension$) (line 9) and copying measure data ($tuple[i, d-1]$) (line 11). The $bits\_of\_dimension$ is calculated according to the values of cardinalities at line 3. The number of bytes for each compressed tuple is ($\lceil \frac{bits\_of\_dimension}{8} \rceil +$ *measure data size*). The number of tuples in the current block is then determined at line 5: $\lceil \frac{block\_size - header\_length}{number\ of\ bytes\ for\ each\ tuple} \rceil$. The tuples are compressed one by one, and a new block is created when the current block is full (line 13). Because of the measure data's various formats, the measure data are stored by bytes. A few bits of gap between the dimensional data and measure data may be generated in each compressed tuple.

Algorithm 9 shows the details of the BIT algorithm with bit compaction by using our proposed data structure. In every compressed tuple, instead of appending measure data right after the compressed dimensional data, the measure data are copied to the measure data area of the compressed block (line 12). All compressed dimensional data are compacted in bits and placed together (line 10). The number of tuples is

Algorithm 8 BIT Data Cube Compression Algorithm (Without Bit Compaction)

Input: Cardinalities ($C[i]$) for each attribute domain ($A[i]$);
A d-dimension view to be compressed.
Output: The compressed view,out_buf.

```
1: bits_of_dimension = 0;
2: for i = 1 to dimension do
3:    bits_of_dimension+ = ⌈log2(C[i])⌉;
4: end for
5: Compute number of tuples for current block;
6: Create a block-header;
7: for i = 0 to number_of_tuples do
8:    for j = 1 to dimension do
9:       Compact tuple[i, j] into log2(C[j]) bits and append it to out_buf;
10:   end for
11:   Copy tuple[i, d − 1] to out_buf;
12:   if current block is full then
13:      Create a new block header
14:   end if
15: end for
```

computed as $\lceil \frac{block\_size - header\_length}{(bits\_of\_dimension + measure\_size * 8)} \rceil$ at line 5. Line 7 computes the offset of the measure data area: $measure\_offset = \lceil \frac{bits\_of\_dimension * number\_of\_tuple\_per\_block}{8} \rceil$.

Algorithm 10 presents the process of decompressing a block of compressed view data coded by using Algorithm 8. The number of tuples in the current block is retrieved from the block header (line 1). Line 5 computes the number of bytes for each compressed tuple according to the cardinality information (line 3). The dimensional data is converted from bit format at line 8 and the measure data is simply copied from the following address (line 10). A whole compressed view is decompressed block by block.

By using the XTDC compacted data structure, Algorithm 11 gets the measure data from the measure segment of the compressed block (line 10). Line 5 computes the offset of measure data area according to the cardinality information (line 3) and

Algorithm 9 BIT Data Cube Compression Algorithm (With Bit Compaction)

Input: Cardinalities ($C[i]$) for each attribute domain ($A[i]$)
A d dimension view to be compressed.
Output: A compressed view, *out_buf*, in XTDC compact format.

```
1: bits_of_dimension = 0;
2: for i = 0 to dimension − 1 do
3:    bits_of_dimension+ = log2(C[i]);
4: end for
5: Compute number of tuples for current block;
6: Create a block_header
7: Compute measure_offset;
8: for i = 0 to number_of_tuples do
9:    for j = 0 to dimension − 1 do
10:      Compact tuple[i, j] into log2(C[j]) bits and append it to out_buf;
11:   end for
12:   measure_offset[i] = tuples[i, d − 1];
13:   if current block is full then
14:      Create a new block header
15:      Compute measure_offset;
16:   end if
17: end for
```

Algorithm 10 BIT Data Cube Decompression Algorithm (Without Bit Compaction)

Input: A block of compressed data: *in_buf*;
Cardinalities ($C[i]$) for each attribute domain ($A[i]$).
Output: Decompressed data: *out_buf*

```
1: number_of_tuples = in_buf[0];
2: for i = 0 to dimension − 1 do
3:    bits_of_dimension+ = log2(C[i]);
4: end for
5: bytes_of_dimension = ⌈bits_of_dimension/8⌉;
6: for i = 0 to tuples_per_block do
7:    for j = 0 to dimension − 1 do
8:       out_buf[i, j] = convert_to_integer(log2(C[j]), in_buf);
9:    end for
10:   out_buf[i, dimension] = in_buf[bytes_of_dimension];
11:   in_buf+ = bytes_of_dimension + bytes_of_measure;
12: end for
```

header information (line 1).

Algorithm 11 BIT Data Cube Decompression Algorithm (With Bit Compaction)

Input: A block of compressed data:*in_buf*;
Cardinalities ($C[i]$) for each attribute domain ($A[i]$)
Output: Decompressed data, *out_buf*

```
1: number_of_tuples = in_buf[0];
2: for i = 0 to dimension − 1 do
3:    bits_of_dimension+ = log2(C[i]);
4: end for
5: Compute measure_offset;
6: for i = 0 to tuples_per_block do
7:    for j = 0 to dimension − 1 do
8:       out_buf[i, j] = convert_to_integer(log2(C[j]), in_buf);
9:    end for
10:   out_buf[i, dimension] = measure_pointer[i];
11: end for
```

As we described in Chapter 2, the BIT compression technique is attribute-level compression. The smallest piece of compressed data, which can be located without touching (decompressing) any other data, is a single attribute. Further more, all the tuples in the entire view are compressed into the same format and hold the same length of bytes.

5.2.2 TDC Data Cube compression

We also apply the TDC [37] database compression algorithm achieve data cube compression in PANDA. The implementation follows the fundamental ideas of [37] except that we use our tuple computation algorithms, *tuple_minus* and *tuple_add*, to avoid data overflow during the computation of tuple differences . As proposed by [37], TDC statically partitions tuples by block, which contains a fixed number of tuples. Algorithm 12 shows the details of the implementation. The block header, which contains the number of tuples (*tuples_per_block*) and the uncompressed attribute values of

the first tuple, can be created before the compression (line 3). Line 5 computes the difference of the consecutive tuples by using *tuple_minus* to reduce the risk of data overflow. The difference values are stored in integer format (line 6) and the measure data is copied afterward (line 7).

Algorithm 12 TDC Data Cube Compression Algorithm

Input: A view to be compressed: *in_buf*
Meta data of the view,
number_of_tuples, *tuples_per_block*
Output: Compressed view: *out_buf*

```
1: processed_tuples = 0;
2: while processed_tuples < number_of_tuples do
3:    Set block header;
4:    for i = 1 to min(tuples_per_block, number_of_tuples − processed_tuples) − 1
      do
5:       difference = tuple_minus(tuple[i], tuple[i − 1]);
6:       out_buf[header_length + 2(i − 1)] = difference;
7:       out_buf[header_length + 2i − 1] = in_buf[i, dimension − 1];
8:    end for
9:    processed_tuples+ = tuples_per_block;
10: end while
```

Algorithm 13 decompresses a view compressed in TDC format, which is the opposite computation of Algorithm 12. The number of tuples per block and the value of the first tuple of the current block are retrieved from the block header (line 3, 4). The tuple is decompressed by adding the difference to its preceding tuple using *tuple_add* (line 8).

In the implementation of the above two algorithms (BIT, TDC), the length of each compressed tuple is known in advance, so we can predict the number of tuples that can be compressed in one block and create the block header in advance.

Algorithm 13 TDC Data Cube Decompression Algorithm

Input: Compressed view: *in_buf*
Meta data of the view;
number_of_tuples;
Output: Decompressed view:*out_buf*

```
1: processed_tuples = 0;
2: while processed_tuples < number_of_tuples do
3:   tuples_per_block = in_buf[0];
4:   tuple[0] = get_first_tuple_from_header();
5:   out_buf[processed_tuples] = tuple[0];
6:   for i = 1 to tuples_per_block do
7:     difference = in_buf[header_length + 2(i − 1)]);
8:     out_buf[processed_tuples + i] = tuple_add(tuple[i − 1], difference);
9:     output[processed_tuples+i, dimension−1] = in_buf[header_length+2i−1];
10:  end for
11:  processed_tuples+ = tuples_per_block;
12:  in_buf+ = header_length + 2 ∗ tuples_per_block;
13: end while
```

5.3 Applying a Conventional Compression Library

In order to compare the performance between the XTDC data cube compression technique and a conventional one, we plug an Open Source conventional compression library, BZIP [45, 6], into the PANDA system. We choose maximal buffer size, 900KB, plus the size of the output to achieve the highest compression ratio that BZIP could reach.

As described in Section 5.1, the compression libraries, *BZ2_bzBuffToBuffCompress*, is called in the compression phase before the physical writing. The decompression library, *BZ2_bzBuffToBuffDecompress*, is called in the decompression phase after physical reading. Since BZIP is a global compression technique, no meta data, such as cardinality of the dimensions, are used during the compression process.

5.4 Conclusion

In this chapter, we introduced the strategy of plugging the Compression Interface into the PANDA's I/O Manager in order to compress and decompress data cubes. By doing so, we allow the XTDC algorithms to be integrated into PANDA without touching other modules in the system. In order to compare the compression efficiency between XTDC and the existing database compression techniques, we implemented the TDC and the BIT database algorithms, as well as the BZIP data compression algorithm. The evaluation will be discussed in the next chapter.

Chapter 6

Evaluation

This chapter discusses the performance of data cube compression techniques implemented in the previous chapters. 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 and compression/decompression speed.

Compression ratio The most popular method of measuring the performance of a compression technique is the compression ratio. It is computed by dividing the original number of bytes by the number of bytes remaining after data compression is applied [25]. 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),

$$CR = \frac{dimension\ size\ without\ compression}{dimension\ size\ with\ compression} \tag{6.1}$$

The total compression ratio, $total\_CR = \frac{data\ size\ without\ compression}{data\ size\ with\ compression}$, is also used to evaluate the compression performance in some cases. Given the compression

ratio (CR), the expected data size reduction is given by REDUC = $(1-1/CR)*$ 100% [42].

Compression speed and decompression speed To compute a view in PANDA with data cube compression, the parent node (view) of it is read and decompressed, and then the target view is compressed before being written to disk. 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.

The memory overhead, which is also important in terms of the scalability of compression algorithms [47], was discussed in the previous chapter. There is no extra memory needed for the BIT and TDC data cube compression implementations. An extra data buffer of constant size is used in the XTDC algorithm to compact the data at the bit level.

We divide our evaluation into three major parts. Section 6.2 evaluates the compression efficiency, both in terms of compression ratio and compression time, for single view computation. Section 6.3 offers the same analysis, but does so in the context of a fully materialized PANDA data cube. In both sections, we provide a performance comparison of the BIT, TDC and XTDC data cube compression algorithms, as well as a conventional data compression algorithm, BZIP. 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 2 GHz Intel Xeon processor, 1.5GB RAM, and 60 GB IDE disks.

The third component of the analysis, presented in Section 6.4, looks at the application of the Hilbert Space Filling Curve technique to the XTDC algorithms. These experiments focus specifically on single view compression ratios. Finally, in Section 6.5, we show some additional tests related to our discussion.

With respect to time oriented tests, such as the full cube pipesort comparison shown in Figure 6.5, the running time is the average of three timed runs using the same input datasets. For the compression ratio evaluation tests, the results depend on the meta data associated with the views, ie. cardinalities, dimensions and number of tuples. All of the output data is also decompressed by using the same module that is plugged into the IO Manager and is compared with the results obtained from original system.

6.1 Test Cases

Real data warehouse systems usually contain between 4 and 15 dimensions [30]. 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 feature attributes (dimensions), and the number of unique values (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

Table 6.1: 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

Table 6.2: The cardinalities of view ABCDFJG

Dimension	0	1	2	3	4	5	6
Name of Dimension	A	B	C	D	F	J	G
Cardinality	6	10	50	8	12	16	3

Size (1M), b) Dimension Count (10). The meta data, which is a group of arbitrary cardinalities that is used by all of our test cases, is listed in Table 6.1.

The PANDA system obtains the meta data (the number of dimensions and the cardinality of each dimension) of a single view according to the name of the view. Example 6.1.1 shows how this works.

Example 6.1.1 *Get the meta data of view ABCDFJG. There are 7 dimensions and 1 measure field in view ABCDFJG. The cardinality of each dimension is computed according to its name and its corresponding cardinality. For instance, this view's first dimension is named 'A', and its corresponding cardinality, defined in Table 6.1, is 6. Table 6.2 lists all cardinalities for each dimension of this view.*

As we discussed in the previous chapter, our implementation allows users to select the block size in multiples of 4KB, in order to adapt to various file systems. The block size is 8KB in all of our tests.

6.2 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,

Table 6.3: 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

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 6.3.

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 [45] 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.

Figure 6.1 compares the compression ratios attained by applying data cube compression techniques and global compression libraries (tools) to the views listed in Table 6.3. 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. In fact, the 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 since TDC 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

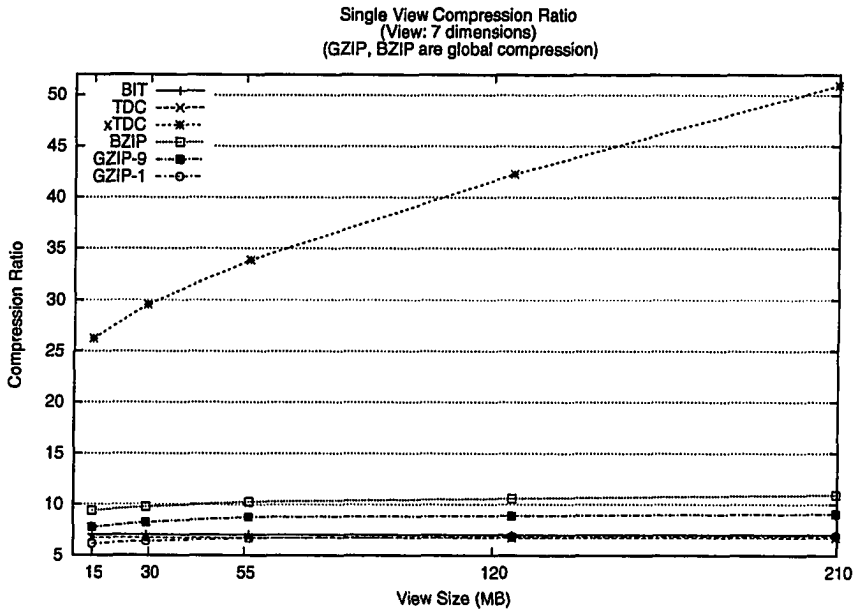

Figure 6.1: Compression ratio comparisons for single view compression

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. With the same dimension cardinalities, the higher the number of tuples in a view, the smaller the maximal difference in that block might be, and the the greater the likelihood of consecutive 1-differences. 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 6.2 presents the running 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 6.3 compares the total time (compression time plus decompression time) for the different techniques.

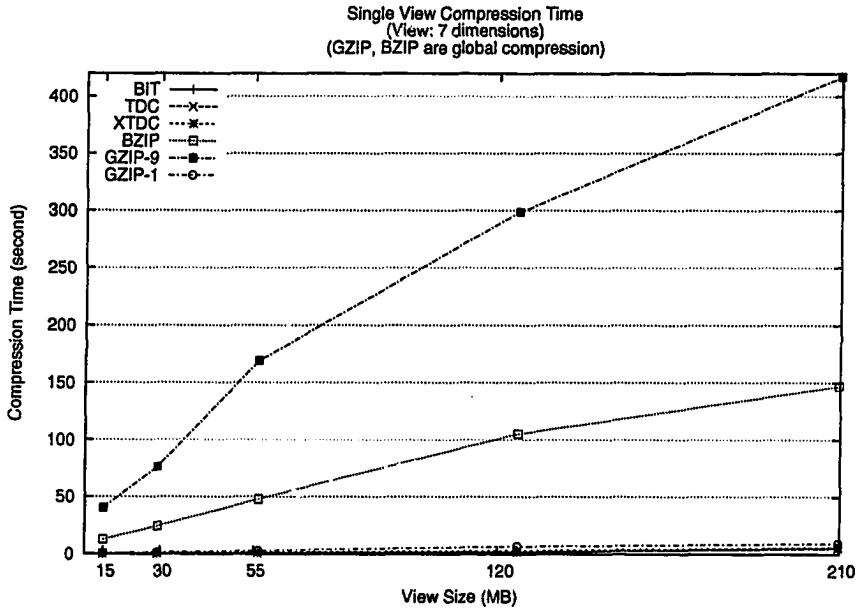

Figure 6.2: Compression time comparisons for single view compression

6.3 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 6.4 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 6.1. Consistent with the result in Figure 6.1, XTDC reaches a much higher compression ratio than the others (BIT, TDC) do.

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

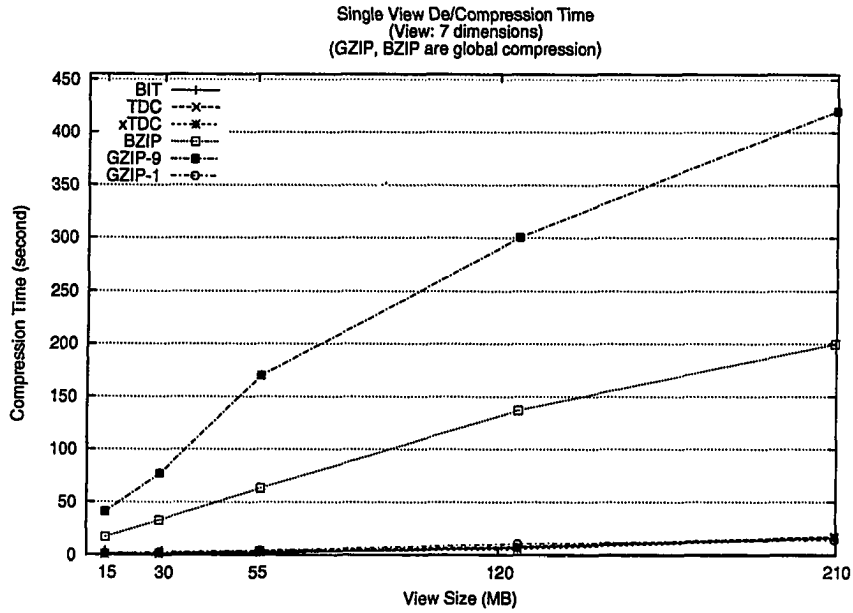

Figure 6.3: Total runtime (compression and decompression) comparison for single view compression

a 29.4 to 1 compression ratio, which reduces the dimensional data from 9778 MB to 333MB. Figure 6.4 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 6.2. 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 the compression ratios of BIT and TDC are not significantly affected as the number of

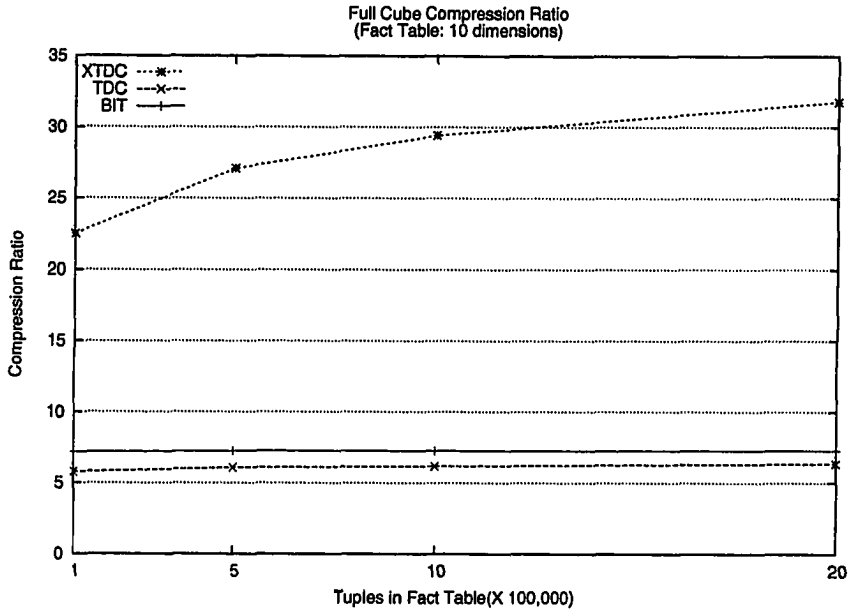

Figure 6.4: The comparison of full cube compression ratios

tuples increases. Since the BIT algorithm uses bit form to express each attribute, the compression ratio only depends on the cardinalities of the dimensions. With respect to the TDC algorithm, the tuple difference value is represented as an integer — four bytes in our implementation. The compression ratio depends on the number of dimensions instead of number of tuples.

Figure 6.5 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

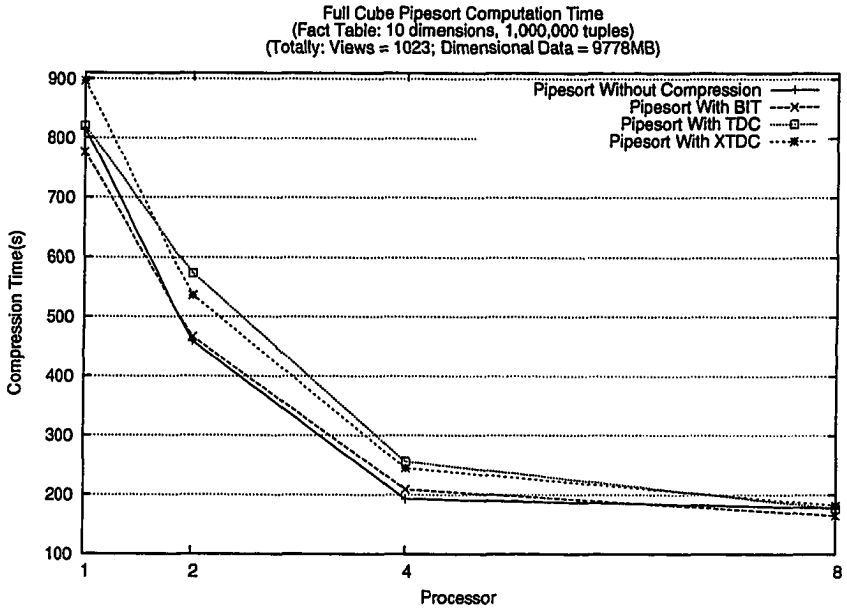

Figure 6.5: Runtime for parallel full cube generation with compression

compression ratio is 7.7 to 1. As we can see from Figure 6.4 and Figure 6.5, 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.

6.4 XTDC with Hilbert Orders

This section focuses on demonstrating the possibility of using the Hilbert Space Filling Curve technique in the XTDC data cube compression algorithms. Since the total compression ratio is determined by the compression ratio of each single sub view, we only evaluate the results of compressing single views applying Hilbert ordering. We use Doug Moore's "Fast Hilbert Curve Generation" code [35] which takes the base-2 logarithm of the largest dimension cardinality as the *order* by which to build the

Hilbert curve computation space. In order to compare the performance between using mix-radix ordering in XTDC and using Hilbert ordering in XTDC, all test views in this section are generated with a fixed cardinality, 32, for all dimensions. By doing so, the two ordering methods are running in the same searching spaces.

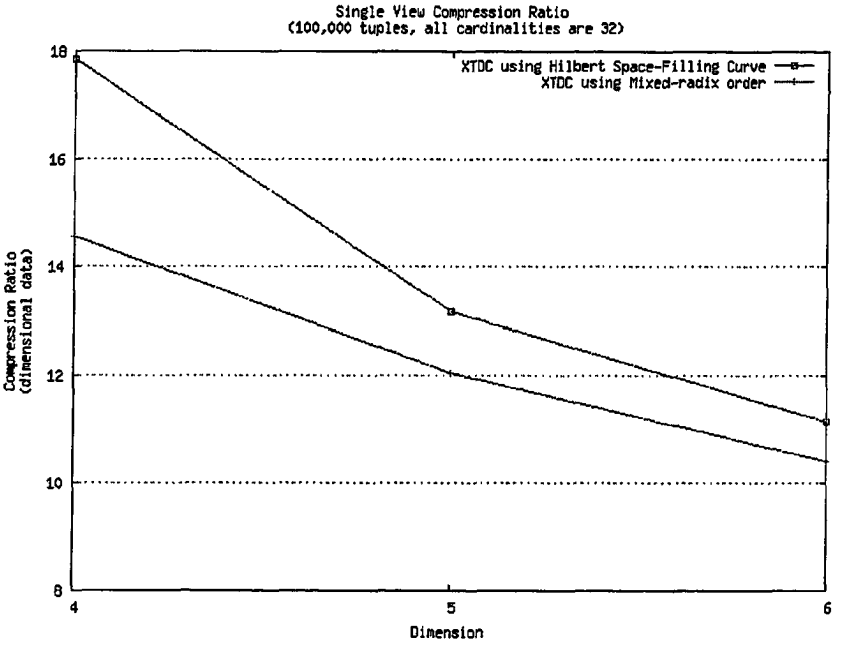

Figure 6.6: The comparison of compressing single views with different dimensions

Figure 6.6 demonstrates that using the Hilbert space filling curve, XDTC reaches high ratios in single view compression. As we discussed before, the value range of the differences between conjunctive tuples affects the compression ratio when using the XTDC algorithms. The fewer bits required to store those differences, the higher the compression ratio is.

In this experiment, the ratio curves decline when the number of dimensions increases. This doesn't necessarily mean that the XTDC compression ratio decreases

when the number of dimensions goes higher. In fact, our previous experiments demonstrate that XTDC is more efficient when the number of dimensions and tuples of the view are larger. Instead, it happens because our test views have much fewer tuples than our previous experiments. As the number of dimensions increase, the computation space becomes bigger, thus the range of difference values between consecutive tuples is larger. Therefore, XTDC needs more bits to represent one tuple. Restricted by Moores's Hilbert order generation code (the cardinalities have to be power of 2 in order to reproduce the same search space as mix-radix order), the test views are not as large as in our previous experiments. So the compression ratios are not as high as we might otherwise expect. Even so, the compression ratios are much higher than the 7:1 ratio achieved by BIT and TDC.

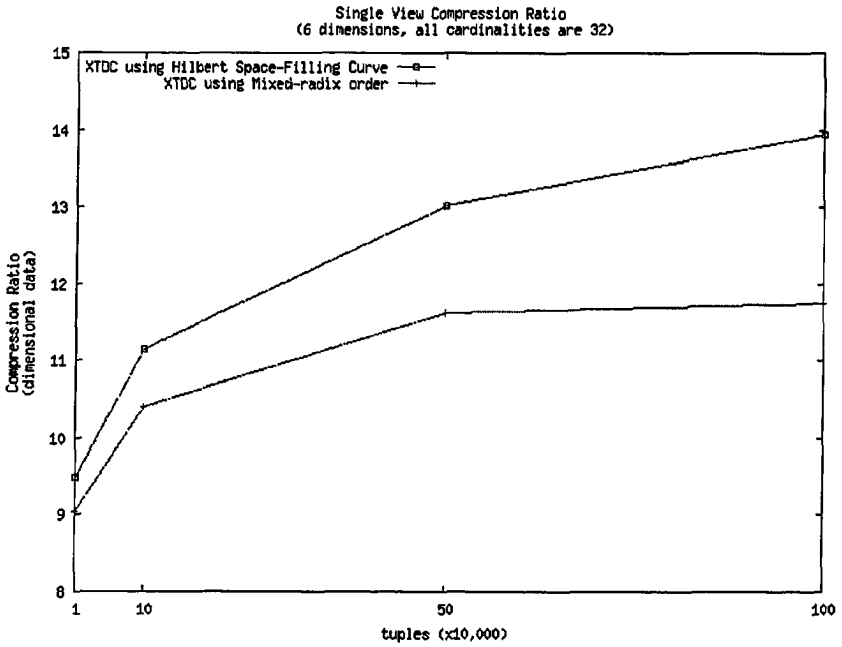

Figure 6.7: The comparison of compressing single views with different number of tuples

Figure 6.7 shows the compression efficiency when compressing views with varying tuple counts. Again, in the same computation space, the more tuples, the smaller the difference values are. The ratio curves go up when the number of tuples increases.

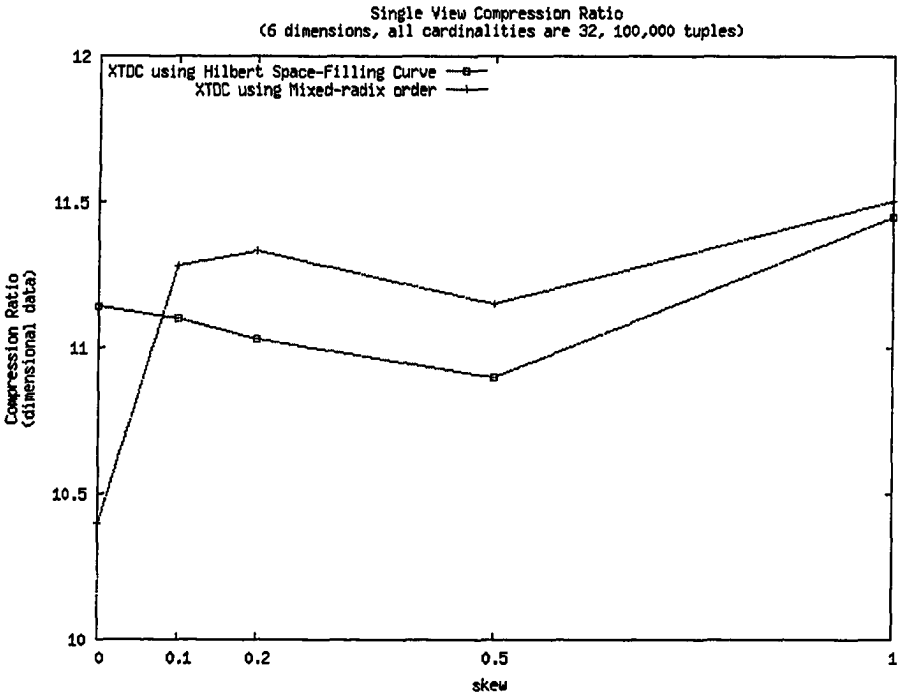

Figure 6.8: The comparison of compressing single views with different skew

The Hilbert Space Filling Curve groups spatially related points into nearby positions on a one-dimensional line. In environments containing skewed data (i.e., points arranged in clusters) we would expect the range of difference values to be smaller when using Hilbert ordering since, by definition, Hilbert curves pack nearby points closely together. Specifically, the existence of skew reduces the distance between points, leading to potentially better compression. Figure 6.8 presents the compression ratios for data sets that have been created with varying degrees of skew. We note that when using the Data Generator to create the experimental data, skew is produced with a *zipfian* function [16]. In this case, *zipf*=0 corresponds to no skew, *zipf*=0.5

to moderate skew, *zipf*=0 to heavy skew. The experiment shows that the XTDC with Hilbert ordering maintains a stable compression ratio when the skew changes. However, it is still too early to conclusively say that XTDC is more efficient when the skew goes up (as we expect). One of the main reasons is that XTDC does not utilize knowledge regarding the data distribution, which is a potential improvement of XTDC in our future work.

The experiments show that applying the Hilbert space filling curve technique to our XTDC algorithm can produce high data cube compression ratios. A typical compression ratio of a single view with 24 MB dimensional data is 13.9 to 1. Because of the significant properties of Hilbert curve — in terms of its ability to dramatically improve the quality of range query resolution — our XTDC data cube compression technique holds huge potential to be used in multi-dimensional indexing applications.

6.5 Additional Tests

This section presents additional experiments that demonstrate the benefit of using our proposed data structure and the counter mechanism. As we discussed in Section 4.3, one of the primary objectives of using the compacted data structure is to remove spare bits (gaps) caused by byte-alignment. In the BIT technique, each compressed attribute uses a fixed number of bits. The compression ratio is dictated by the cardinalities of the dimensions rather than the number of tuples.

The first experiment chooses a group of single views generated from fact tables with 10^5 tuples. The cardinalities of each dimension are the same as listed in Table 6.1. The number of dimensions in each view is from 6 to 10. Table 6.4 shows meta data for these testing views. Figure 6.9 shows compression ratio gains made by compacting compressed dimensional data. As the results show, the more bits of

Table 6.4: The number of bits per tuple of testing views

#Dimension	Views Name	Bits per tuple	Bits of Gap
6	ABCDEF	25	7
7	ABCDEFG	27	5
8	ABCDEFGH	31	1
9	ABCDEFGHI	34	6
10	ABCDEFGHIJ	39	1

gap each compressed view has, the greater the increase in compression by using the compacted data structure.

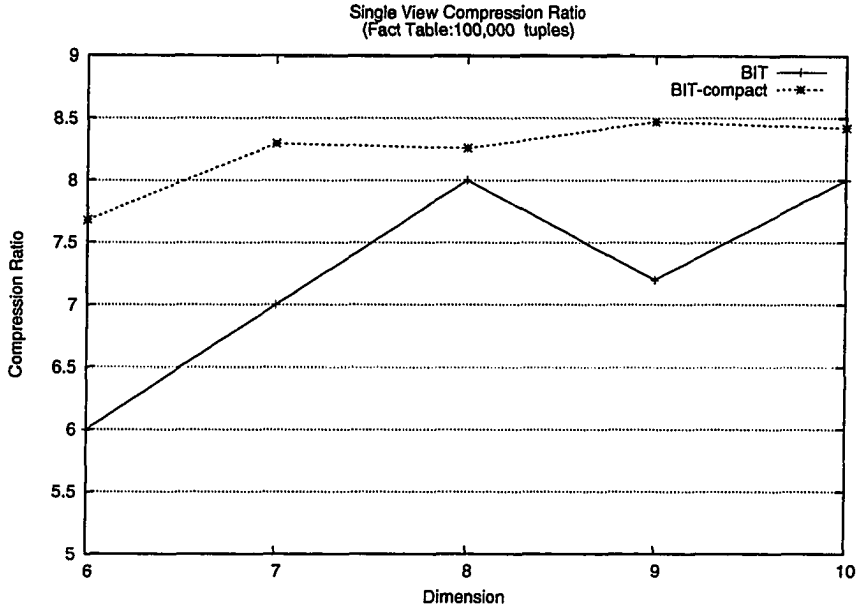

Figure 6.9: The comparison between BIT techniques with/without bit compaction.

Previous experiments have demonstrated the benefits of expressing tuple differences in bits and of using our data structure to compact these differences together using the XTDC technique. The counter mechanism is another technique we proposed in the XTDC algorithm to pursue a high compression efficiency for those views that have low dimensions and have relatively small cardinalities compared with the

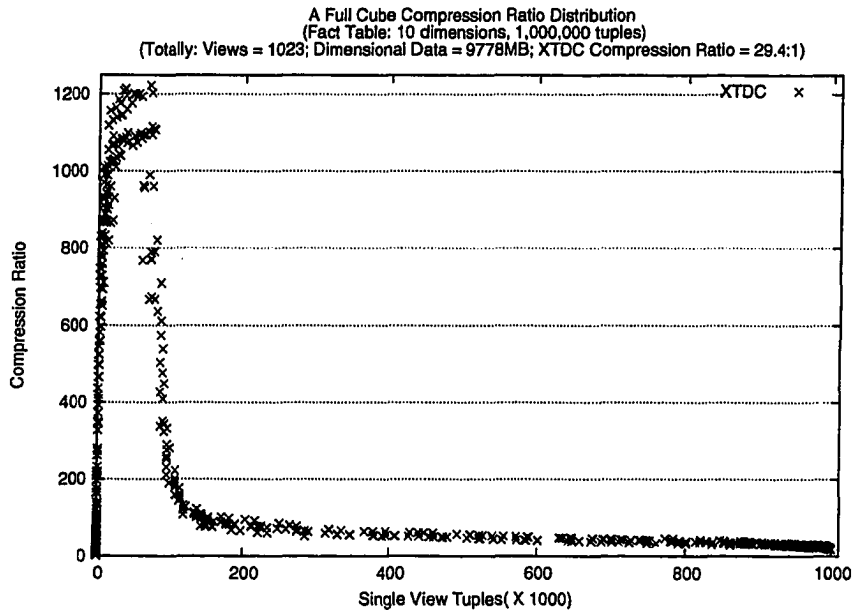

Figure 6.10: A distribution of XTDC compression ratios in a full cube computation

number of tuples. Figure 6.10 presents a distribution of XTDC compression ratios in a full cube computation. The fact table has 10^6 tuples and 10 dimensions. In total, 1023 views are created in the full cube computation. The size of the dimensional data is reduced from 9778MB to 333MB. The compression ratio for the entire cube is 29.4 to 1. The space reduction is 9445MB (96.6%). In general, the views in which the number of tuples is less than 4×10^5 have lower dimension counts. So for these views, the possibility of consecutive 1-differences with consecutive tuples is somewhat higher. In these cases, the counter mechanism is able to produce compression ratios on individual views that exceed 1000.This trend stabilizes between 16 and 60 for views that have more than 4×10^5 tuples. We note, however, that the curve's decline does not necessarily mean that the success of the XTDC algorithm deteriorates with larger volume data sets. Rather, the reason for the decline is simply that views with

a higher number of dimensions receive less benefit from the counter. In fact, as Figure 6.1 shows, for views that have the same number of dimensions and cardinalities, the larger the number of tuples, the higher the compression ratio for XTDC.

6.6 Conclusion

In this chapter, we evaluated the XTDC algorithms in the parallel OLAP computation system — PANDA. The experimental results show that XTDC reaches much higher compression ratios than the existing compression algorithms in both single view compression and full data cube compression. The typical compression ratio is 50.9 to 1 for a single view with 7 dimensions and 10^7 tuples, and 31.8 to 1 for a full cube with a fact table of 10 dimensions and 2×10^6 tuples. The results also demonstrate that XTDC is more efficient when the data cube grows in size — in terms of a larger number of tuples and dimensions in the fact table. The experimental results also show that the Hilbert Space Filling Curve technique is well suited to the XTDC algorithms.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis proposes an efficient data cube compression algorithm — XTDC — and demonstrates its effectiveness on a mature parallel OLAP computation system. By building upon a number of existing compression algorithms, and by exploiting the fundamental characteristics of the data cube storage model, we have implemented and tested the following four compression techniques in the PANDA environment:

1. The XTDC algorithms — with the corresponding data structure. Two efficient tuple differential computing algorithms, *tuple_add* and *tuple_minus*, are proposed when using mix-radix ordering. The Hilbert curve ordering technique is also implemented on top of the XTDC algorithms;
2. The BIT tuple compression algorithms;
3. The TDC block-wise compression method;
4. BZIP, the conventional global data compression technique.

The experimental results demonstrate that the XTDC technique achieves much higher compression ratios than would be the case by simply applying existing database compression techniques and conventional data compression algorithms to data cube computation. 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. The dimensional data with large dimensions can be encoded into a few bits.

2. Bit compaction: The tuple differences are stored in bit form and are compacted. The encoded tuples are stored in bit format to save each single bit of space.

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: For those tuples that can fit in one block, these two kinds of data are stored separately in the same block. The dimensional data can then be compressed and compacted together to get rid of the gaps caused by byte-alignment.

5. The counter mechanism: In the large full cube construction, some views have relatively low dimensions but a large number of tuples. The differences of the conjunctive tuples are likely to be 1's. We use a counter, instead of wasting storage space on a set of records, to represent those tuples at the beginning of the block whose difference values are just consecutive 1's.

6. Using meta data information: Knowledge of the data cube is used when compressing and decompressing. For example, we use it when calculating the tuple

differences and organizing the compressed blocks.

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 compression ratio of single views reaches as high as 1353 to 1 in some of our tests. 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.

We acknowledge that "real world" application data may differ from our experimental data, in terms of the cardinalities and skew of tuples. Of particular concern is the fact that as the cardinalities increase, more bits may be needed to store the differences of tuples, thus drawing down the compression ratio. However, in data warehousing applications, the dimensional data are usually surrogate keys, which are stored as consecutive integers. The values of conjunctive tuples are therefore often quite similar. XTDC still can use the tuple differential coding to pursue high compression ratios.

With respect to skew of tuples in real data sets, as we discussed in Section 6.4, we would expect high compression ratios for those blocks that contain only one cluster of tuples, since their differences tend to small, but low compression ratios for those blocks that contain more than one cluster of tuples, because the differences of conjunctive tuples that belong to different clusters are likely bigger.

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. Since all information about tuples is encoded in individual blocks, the data cube operation can be done when the data cubes are still compressed. This thesis proposes two algorithms for random access and sub cube generation based on compressed data. The purpose is to allow for the possibility of manipulating the compressed data cube without decoding the whole views, thereby improving OLAP computing performance.

The experimental results also show 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 multi-dimensional indexing.

7.2 Future Work

The research described in this thesis provides the foundation for a space and storage efficient data cube compression technique. Since this 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. Below we identify a number of these possibilities:

1. Indexing compressed data blocks. As noted in the previous chapter, the compressed data cubes are sorted and organized at the block level. The first tuple of a block is stored in the block header in uncompressed format. It is very convenient to build an index for fast random tuple location. Since the number of blocks is reduced for the compressed view, the size of the index file is significantly decreased as well.

2. Computing compressed data cubes. In our current implementation, an XTDC

interface is plugged into PANDA to compress and decompress data during the physical I/O phases. PANDA computes views using uncompressed data in main memory. Section 4.5 discussed a technique to generate compressed sub views from a compressed view. By doing so, we will not only save main memory space during data cube computation, but also avoid most of the data compression and decompression processes. There could be a significant performance improvement gained by implementing such computation in PANDA.

3. Applying the Hilbert Space Filling Curve technique. Our experimental results show that Hilbert curve ordering can be applied to the XTDC algorithms to achieve very high compression ratios when the Hilbert ordering has the same computation space as the mix-radix ordering does. However, since the Hilbert order generation code we are using takes the largest cardinality of the dimensions of the view to determine its computation space, it is not efficient to simply apply this code to compress data cubes in practical OLAP computation systems. We would like to improve the Hilbert order generation code and apply it to the PANDA system. By doing so, both data cube compression and multidimensional indexing can share the full benefit of Hilbert ordering.

Bibliography

[1] Christopher Adamson and Michael Venerable. *The Data Warehouse Design Solutions.* John Wiley & Sons, Inc, 1998.

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

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

[4] Nieves R. Brisaboa, Eva L.Iglesias, Gonzalo Navarro, and Jose T.Parama. An efficient compression code for text databases. In *ECIR*, volume 2633 of *Lecture Notes in Computer Science*, pages 468–481. Springer-Verlag, 2003.

[5] Adam L. Buchsbaum, Donald F. Caldwell, Kenneth Ward Church, Glenn S. Fowler, and S. Muthukrishnan. Engineering the compression of massive tables: an experimental approach. In *Symposium on Discrete Algorithms*, pages 175–184, 2000.

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

[7] Ying Chen, Frank Dehne, Todd Eavis, and Andrew Rau-Chaplin. Parallel ROLAP data cube construction on shared-nothing multiprocessors, 2002.

[8] Ying Chen, Frank Dehne, Todd Eavis, and Andrew Rau-Chaplin. Building large ROLAP data cubes in parallel. In *IDEAS*, pages 367–377, 2004.

[9] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In *SIGMOD Conference*, 2001.

[10] Zhiyuan Chen and Praveen Seshadri. An algebraic compression framework for query results. In *ICDE*, pages 177–188, 2000.

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

[12] Frank Dehne, Todd Eavis, Susanne Hambrusch, and Andrew Rau-Chaplin. Parallelizing the data cube. *International Conference on Database Theory*, 2002.

[13] Frank Dehne, Todd Eavis, and Andrew Rau-Chaplin. Coarse grained parallel online analytical processing (OLAP) for data mining. *Lecture Notes in Computer Science*, 2074, 2001.

[14] Frank Dehne, Todd Eavis, and Andrew Rau-Chaplin. Computing partial data cubes for parallel data warehousing applications. *Lecture Notes in Computer Science*, 2131, 2001.

[15] Frank Dehne, Todd Eavis, and Andrew Rau-Chaplin. Parallel multi-dimensional ROLAP indexing, 2004.

[16] Todd Eavis. *Parallel OLAP Computing.* PhD thesis, Dalhousie University, 2004.

[17] Travis Gagie. Dynamic length-restricted coding. Master's thesis, University of Toronto, 2003.

[18] William A. Giovinazzo. *Object-Oriented Data Warehouse Design.* Prentic Hall,Inc.,Upper Saddle River, New Jersy, USA, 2000.

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

[20] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *ICDE*, pages 370–379, 1998.

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

[22] Jim Gray and Andreas Reuter. Transaction processing: Concepts and techniques, 1993.

[23] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proceeding ACM SIGMOD Conference*, pages 205–216, 1996.

[24] Gilbert Held and Thomas R. Marshall. *Data Compression.* John Wiley & Sons, New York, NY, USA, third edition, 1991.

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

[26] HPCVL. Website, http://www.hpcvl.org.

[27] X. Huang, H. Lu, and Z. Li. Computing data cubes using massively parallel processors, 1997.

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

[29] Marcus Jurgens. *Index Structures for Data Warehouses*, volume 1859 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002.

[30] Ralph Kimball, Laura Reeves, Margy Ross, and Warren Thornthwaithe. *The Data Warehouse Lifecycle Toolkit.* John Wiley & Sons, Inc, 1998.

[31] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit.* John Wiley & Sons, Inc, second edition, 2002.

[32] Jianzhong Li, Doron Rotem, and Jaideep Srivastava. Aggregation algorithms for very large compressed data warehouses. In *Proceeding of the 25th VLDB Conference*, pages 651–662, 1999.

[33] Abu Sayed Md, Latiful Hoque, Douglas McGregor, and John Wilson. Databases compression using an offline dictionary method. In *ADVIS*, volume 2457 of *Lecture Notes in Computer Science*, pages 11–20. Springer-Verlag, 2002.

[34] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering*, 13(1):124–141, 2001.

[35] Doug Moore. http://www.caam.rice.edu/ dougm/twiddle/hilbert/.

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

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

[38] Panda. Project website, http://www.cs.dal.ca/panda/.

[39] Meikel Poess and Dmity Potapov. Data compression in oracle. In Frederick H. Lochovsky, editor, *Proceedings of the 29th VLDB Conference, Berlin, Germany*, 2003.

[40] Gautam Ray, Jayant R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *International Conference on Management of Data*, 1995.

[41] Kenneth A. Ross and Divesh Srivastava. Fast computation of sparse datacubes. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *Proc. 23rd Int. Conf. Very Large Data Bases, VLDB*, pages 116–125. Morgan Kaufmann, 25–27 1997.

[42] M. A. Roth and S.J.Van Horn. Database compression. *SIGMOD Record*, 22(3), September 1993.

[43] Nick Roussopoulos, Yannis Kotidis, and Mema Roussopoulos. Cubetree: organization of and bulk incremental updates on the data cube. In *Proceeding ACM SIGMOD Conference*, pages 89–99, 1997.

[44] Sunita Sarawagi, Rakesh Agarwal, and Ashish Gupta. On the computating the data cube. *IBM Research Report*, 1996.

[45] Julian Seward. bzip2 and libbzip2, http://sources.redhat.com/bzip2.

[46] James A. Storer. *Data Compression Methods and Theory.* Rockville, Md.: Computer Science Press Inc, 1988.

[47] Peter Wayner. *Compression Algorithms for Real Programmers.* San Diego: Morgan Kaufmann, 2000.

[48] GNU Website. http://www.gnu.org.

[49] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[50] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[51] Wei Biao Wu and Chinya V. Ravishankar. The performance of difference coding for sets and relational tables. *Journal of the ACM (JACM)*, 50(5):665–693, September 2003.

[52] John Yiannis and Justin Zobel. External sorting with on-the-fly compression.

[53] Cui Yu. *High-Dimendional Indexing*, volume 2341 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

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