Efficient OLAP Operations for Spatial Data Using Peano Trees

Baoying Wang Fei Pan Dongmei Ren Yue Cui Qiang Ding William Perrizo
Computer Science Department
North Dakota State University
Fargo, ND 58105
Tel: (701) 231-6257
{baoying.wang, fei.pan, Dongmei.ren, yue.cui, qiang.ding, william.perrizo}
@ndsu.nodak.edu

ABSTRACT
Online Analytical Processing (OLAP) is an important application of data warehouses. With more and more spatial data being collected, such as remotely sensed images, geographical information, digital sky survey data, efficient OLAP for spatial data is in great demand. In this paper, we build up a new data warehouse structure – PD-cube. With PD-cube, OLAP operations and queries can be efficiently implemented. All these are accomplished based on the fast logical operations of Peano Trees (P-Trees*). One of the P-tree variations, Predicate P-tree, is used to efficiently reduce data accesses by filtering out "bit holes" consisting of consecutive 0's. Experiments show that OLAP operations can be executed much faster than with traditional OLAP methods.

Keywords
Online analytical processing (OLAP), spatial data, P-Trees, PD-cube.

1. INTRODUCTION
Data warehouses (DWs) are collections of historical, summarized, non-volatile data, accumulated from transactional databases. They are optimized for Online Analytical Processing (OLAP) and have proven to be valuable for decision-making [9]. With more and more spatial data being collected, such as remote sensed images, geographical information, and digital sky survey data, it is important to study on-line analytical processing of spatial data warehouses.

The data in a warehouse are conceptually modeled as data cubes. Gray et al. introduce the data cube operator as a generalization of the SQL group by operator [3]. The chunk-based multi-way array aggregation method for data cube in OLAP was proposed in [4].

Typically, OLAP queries are complex and the size of the data warehouse is huge, which can cause queries to take very long to complete, if executed directly on raw data. This delay is unacceptable in most data warehouse environments.

Two major approaches have been proposed to efficiently process queries in a data warehouse: speeding up the queries by using index structures, and speeding up queries by operating on compressed data. Many different index structures have been proposed for high-dimensional data [1, 2]. Some special indexing techniques, such as bitmap index, join index and bitmap join index, have been introduced. The bitmap index uses a bit vector to represent the membership of tuples in a table. A significant advantage of bitmap index is that complex logical selection operations can be implemented very quickly by performing bit-wised *and*, *or*, and *complement* operators. However, bitmap indexes are space inefficient for high cardinality attributes. Bitmap indexes are only suitable for narrow domains, especially for membership functions with 0, or 1 as function values. The problem associated with sparse bitmap indexes is discussed in [5]. In order to overcome this problem, some improved bitmaps, e.g., encoded bitmap [7], have been proposed. The join indexes gained popularity from its use in relational database query processing. Join indexing is especially useful for maintaining the relationship between a foreign key and its matching primary keys, from the joinable relations.

Recently, a new data structure, Peano Tree (P-tree) [10, 11], has been introduced for spatial data. P-tree is a lossless, quadrant based compression data structure. It provides efficient logical operations that are well suited for multi-dimensional spatial data. One of the P-tree variations, Predicate P-tree, is used to efficiently reduce data accesses by filtering out "bit holes" which consist of consecutive 0's.

In this paper, we present a new data warehousing structure, PD-cube, to facilitate OLAP operations and queries. Fast logical operations on P-Trees are used to accomplish these OLAP operations. One of the P-tree variations, Predicate P-tree, is used to efficiently reduce data accesses by filtering out "bit holes" consisting of consecutive 0's. Experiments show that OLAP operations can be executed much faster this way than with traditional data cube methods.

This paper is organized as follows. In section 2, we review P-tree structure and its operations. In section 3, we propose a new data warehouse structure, PD-cube, and efficient OLAP operations using P-Trees. Finally, we compare our method with traditional data cube methods experimentally in section 4 and conclude the paper in section 5.

2. REVIEW OF PEANO TREES
A quadrant-based tree structure, called the Peano Tree (P-tree), was developed to facilitate compression and very fast logical operations of bit sequential (bSQ) data [10]. P-Trees can be 1-dimensional, 2-dimensional, 3-dimensional, etc.

The most useful form of a P-tree is the *predicate*-P-tree in which a 1-bit appears at those tree nodes corresponding to quadrants for

* Patents are pending on the P-tree technology. This work is partially supported by GSA Grant ACT#: K96130308.

which the predicate holds. The predicate can be a particular bit-position of a particular attribute or, more generally, a set of values for each of a set of attributes. Pure 1 P-tree (P1-tree) and Non Pure0 P-tree (NP0-tree) are two predicate P-Trees. In Figure 1, a bSQ file with 64 rows is shown, the file is rearranged into 2-D Peano or Z order in the left and the P-Trees (P1-tree and NP0-tree) are on the right. We identify quadrants using a Quadrant identifier, Qid - the string of successive sub-quadrant numbers (0, 1, 2 or 3 in Z or Peano order, separated by "." (as in IP addresses). Thus, the Qid of the bolded and underlined quadrant in Figure 1 is 2.2.

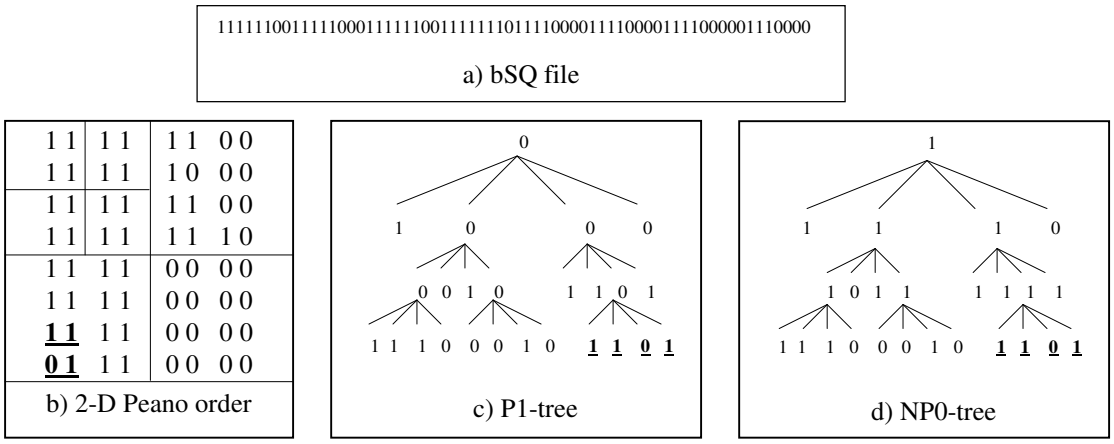


Figure 1. bSQ file, 2-D Peano order bSQ file, P1-tree, and NP0-tree

Logic *and* and *or* are the most important and frequently used P-tree logical operations. By using predicate P-tree operations, we filter "big holes" consisting of consecutive 0's and get the mixed quadrants. Then we only load the mixed quadrants of Peano sequence into main memory, thus reducing the data accesses. The *and* and *or* operations using NP0-tree are illustrated in Figure 2.

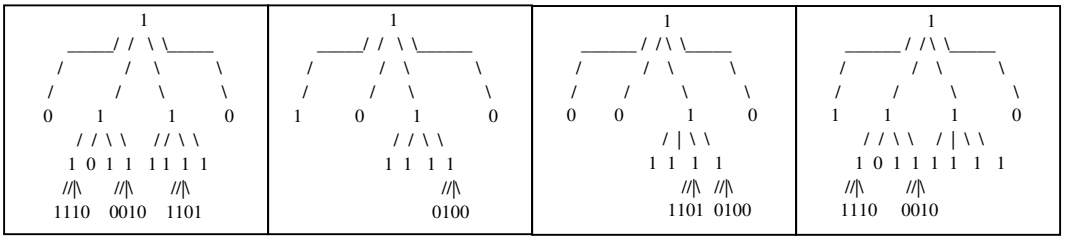


Figure 2. P-tree Operations: and and or

In Figure 2, a) and b) are two NP0-trees, c) and d) are their *and* result and *or* result, respectively. From c), we know that bits at the range of position [0, 32] and [49, 64] are pure 0 since the bits in quadrant with Qid 0, 1 and 3 are 0. Quadrants with Qid 2 are non-pure0 parts. We only need to load quadrants with Qid 2. The logical operation results calculated in such way are the exactly same as anding two bit sequence with reduced data accesses. There are several ways to perform P-tree logical operations. The basic way is to perform operations level by level starting from the root level. The node operation is commutative.

3. OLAP OPERATIONS USING P-TREES

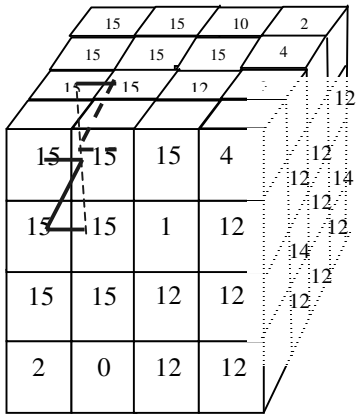
In this section, we first develop a new data warehousing structure, PD-cube, which is the bit-wised representation of data cube. Then we show that OLAP operations, such as slice/dice and roll-up, based on PD-cube, can be efficiently implemented using P-Trees.

3.1 PD-cube

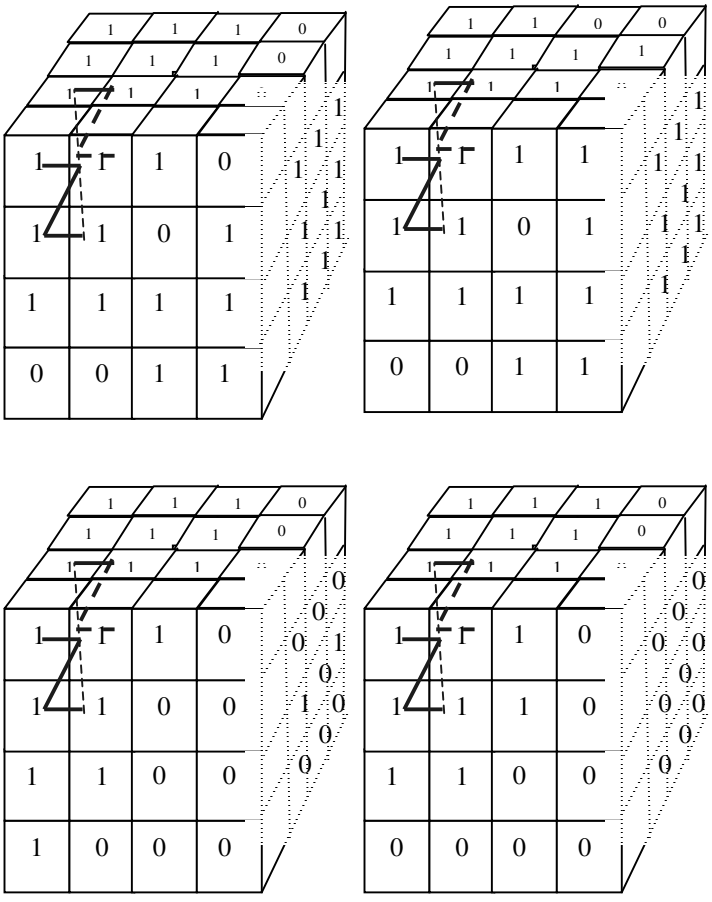
To take advantage of the continuity and sparseness in spatial data, we develop a new data warehouse structure, PD-cube. It is a bit-wised data cube in Peano order which is consistent with P-Trees in terms of data storage. We partition the data cube into bit level in order to facilitate fast logical operations of predicate P-Trees. Predicate P-Trees are tree-based bitmap indexes built on quadrants. By means of fast logical and, or and not operations of predicate P-Trees, PD-cubes achieve efficient data access, which is very important for massive spatial data set.

X	Y	T	Yeild
0	0	0	15 (1111)
1	0	0	15 (1111)
0	1	0	15 (1111)
1	1	0	15 (1111)
0	0	1	15 (1111)
1	0	1	15 (1111)
0	1	1	15 (1111)
1	1	1	15 (1111)
2	0	0	15 (1111)
3	0	0	4 (0100)
2	1	0	1 (0001)
3	1	0	12 (1100)
2	0	1	12 (1100)
3	0	1	2 (0010)
2	1	1	12 (1100)
3	1	1	12 (1100)
0	2	0	15 (1111)
1	2	0	15 (1111)
0	3	0	2 (0010)
1	3	0	0 (0000)
0	2	1	15 (1111)
1	2	1	15 (1111)
0	3	1	2 (0010)
1	3	1	0 (0000)
2	2	0	12 (1100)
3	2	0	12 (1100)
2	3	0	12 (1100)
3	3	0	12 (1100)
2	2	1	12 (1100)
3	2	1	12 (1100)
2	3	1	12 (1100)
3	3	1	12 (1100)
0	0	2	15 (1111)
1	0	2	15 (1111)
0	1	2	15 (1111)
1	1	2	15 (1111)
0	0	3	15 (1111)
1	0	3	15 (1111)
0	1	3	15 (1111)
1	1	3	15 (1111)
2	0	2	15 (1111)
3	0	2	10 (1010)
2	1	2	1 (0001)
3	1	2	14 (1110)
2	0	3	14 (1110)
3	0	3	2 (0010)
2	1	3	12 (1100)
3	1	3	12 (1100)
0	2	2	15 (1111)
1	2	2	15 (1111)
0	3	2	2 (0010)
1	3	2	0 (0000)
0	2	3	15 (1111)
1	2	3	15 (1111)
0	3	3	2 (0010)
1	3	3	0 (0000)
2	2	2	12 (1100)
3	2	2	12 (1100)
2	3	2	12 (1100)
3	3	2	12 (1100)
2	2	3	12 (1100)
3	2	3	12 (1100)
2	3	3	12 (1100)
3	3	3	12 (1100)

(a) Fact Table of Yield in Peano Order



(b) Data Cube of Field Yield



(c) PD-cubes

Figure 3. A Fact Table, its Data Cube and PD-cubes

Figure 3 shows an example of a 3-D data cube representing the crop yield. It has three dimensions: position x (X), position y (Y), and time (T). Since yield is a 4-bit value, we split the data cube of yield into four bit-wised cubes, which are stored in form of predicate P-Trees NP0 and P1.

3.2 OLAP Operations

The OLAP operations are the basis for answering spatial data warehouse questions like: "find all galaxies brighter than magnitude 22", "find average yield of field", or "find the total traffic flow during a period of time." Slice/dice and roll-up operations will be described as follows.

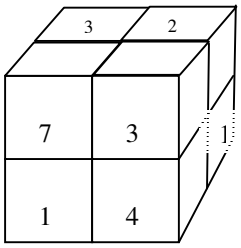
3.2.1 Slice/dice

The Slice operation is a selection along one dimension, while the dice operation defines a sub-cube by performing a selection on two or more dimensions. Slice and Dice are very efficiently accomplished using P-Trees. Typical select statements may have a number of predicates in their "where" clause that must be combined in a Boolean manner. The predicates may include "=", "<" and ">". These predicates lead to two different query scenarios, where "=" clause results in an equal query, and "<" or ">" in a range query.

3.2.1.1 Equal Select Slice

Suppose we have a 3-D data cube representing crop yield with dimensions X, Y and T, where X = {0, 1}, Y = {0, 1} and T = {0, 1}. Attribute "Yield" is converted from decimal to binary. The data cube and its corresponding relational table are shown in Figure 4.

From the table in Figure 4 we build 6 P-Trees as shown in Figure 5 according to the P-tree construction rule. Pi, j, is a P-tree for the jth bit of the ith attribute. For attribute of m bit, there are m P-trees, one for each bit position



X	Y	T	Yield
0	0	0	111
1	0	0	011
0	1	0	001
1	1	0	100
0	0	1	011
1	0	1	010
0	1	1	100
1	1	1	001

Figure 4. A 3-D Cube and Binary Relational Table

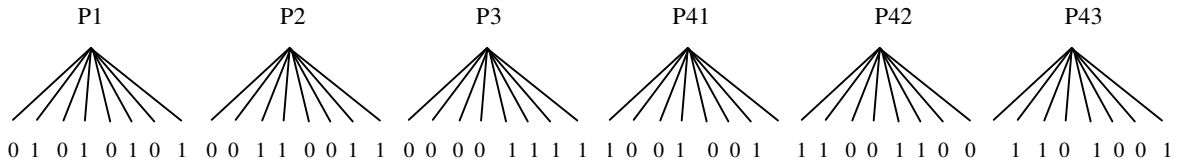


Figure 5. P-Trees for 3-D cube Quadrant

Suppose we want to know that yield at the place where Y = 1. The query is "get yield where Y = 1". The process of selection is: first get P-tree masks, and then trim the P-Trees. The detailed process is discussed as follows.

First, create the P-tree mask, PM = P2. Then use PM to generate a new set of P-Trees. The new P-tree set is the result of selection.

Generation of new P-Trees is simple: just trim the nodes corresponding to the position of pure 0 in PM, and PM is P2 here. Take P43 for example, the trimming process is shown as Figure 6.

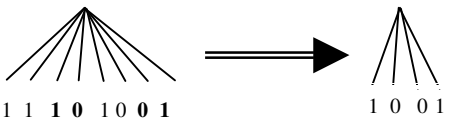


Figure 6. Trim Process of P-tree based on PM

After trimming the whole P-tree set, we can convert the trimmed P-tree set easily to a data cube as shown in Figure 7.

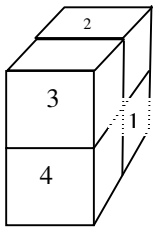


Figure 7. Result Cube of Selection

3.2.1.2 Range Slice

Typical range queries for spatial data warehouse include questions like: "Get crop yield within the range of X > 200 and Y > 150". We analyze three range slice cases as follows.

a. Get yield where Y >1001.

Given the number of bits for the second attribute Y is fixed as 4, we can easily get the values that are greater than 1001. From Figure 8 , we can see that {"Y > 1001"} consists of two subsets {"Y = 11**"} and {"Y = 101*"}, where * is 1 or 0. Therefore the query clause can be written as "where Y = 11** ‖ Y = 101*".

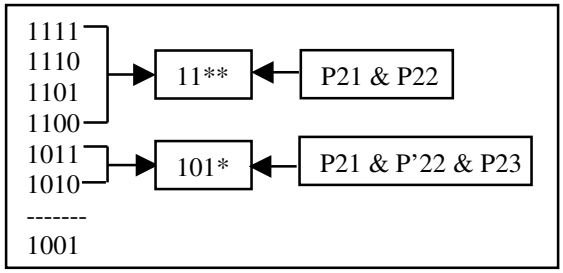


Figure 8. Range Query of the Second Attribute Y > 1001

The query is retrieved by the corresponding P-tree mask PM_{gt} = PM1 ‖ PM2. PM1 is predicate P-tree for data that start with "11" and PM2 is for data that start with "101". Since Y is the second attribute of the Peano Cube, we have

PM1 = P21 & P22
PM2 = P21 & P'22 & P27.

where

P21 – the predicate P-tree for data that has value 1 in the first bit position of the second attribute

P'22 – the predicate P-tree for data that has value 0 in the second bit position of the second attribute

P22, P27 – follows the same rule.

b. Get yield where T > 01011001.

The retrieval process of this query is shown in Figure 9. Attribute T is the third dimension of the PD-Cube. Data set {T > 01011001} can be divided a number of subsets: {"1*******"}, {"011*****"}, {"010111**"} and {"0101101*"}, where * is 1 or 0. The retrieval process can be generalized as follows:

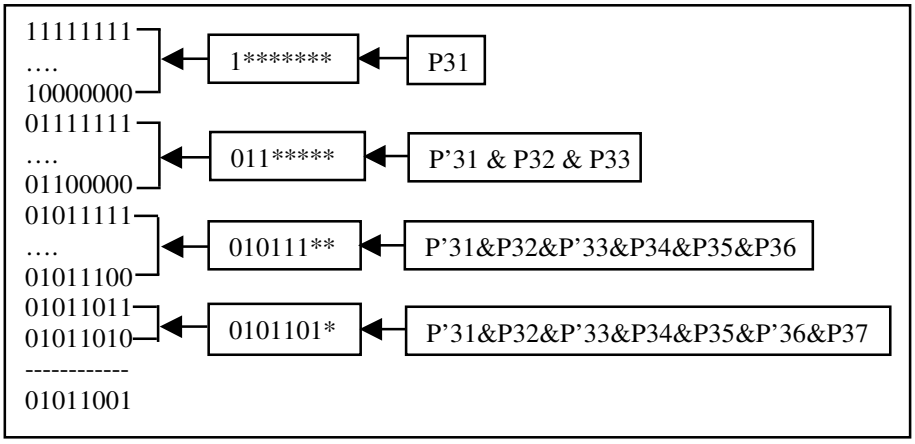


Figure 9. Range Query of the Third Dimension T > 01011001

- The number of subsets is equal to the number of 0 in the bit stream. Data set {T > 01011001} consists of four subsets.
- Start with the highest bit of string until the nth 0, revert it to 1 and append it with wildcard *, we get the first subset. The first subset of {T > 01011001} is {"1*******"}, the second subset is {"011*****"}, and so on.
- Generate mask P-tree for each subset according to bit pattern and position as discussed in previous example. For this example PM1 = P31, PM2 = P31' & P32 & P33, PM3 = P31' & P32 & P33' & P34 & P35 & P36 and PM4 = P31' & P32 & P33' & P34 & P35 & P36' & P37.
- AND all mask P-Trees together to get a general mask P-tree. For this example PM_{gl} = PM1 ‖ PM2 ‖ PM3 ‖ PM4.

c. Combination of Equal Query and Range Query

It is often the case that "where" clause has ">=". For example, "get yield where T >= 01011001". We can easily divide data set {"T >=01011001"} into two subsets, {"T >01011001"} and {"T = 01011001"}. Therefore the mask P-tree $PM_{ge} = PM_{gt} ‖ PM_{eq}$. PM_{gt} is related to range query and PM_{eq} is related to equal query.

For "where" clause like "01111011 > T > 01011001", we also need to divide set {"01111011 > T > 01011001"} into two subsets, {"01111011 > T"} and {"T > 01011001"}. The mask P-tree $PM_{gl} = PM_{gt} ‖ PM_{lt}$.

d. Complement of Range Query

We have just analyzed "where" clause with ">" or ">=". But we also want "<" or "<=". For example, "get yield where T <= 01011001." Obviously, data set {"T <= 01011001"} is the complement of {"T > 01011001"}. With the result of query "Get yield where T > 01011001, we can easily retrieve query "Get yield where T <= 01011001" by making complement, i.e. $PM_{le} = PM'_{gt}$.

Dice operation is similar to slice operation. Only dice involves more than one dimension while slice is on one dimension. The example of dice can be "get yield where T <= 01011001" & "Y > 1001." We just need to *and* P-tree masks of slice on dimension T and Y to get the final P-tree for dice.

3.2.2 Rollup

PD-cube is stored in Peano order rather than in raster order. Therefore, the rollup of PD-cube is accomplished differently from the traditional data cube as a result of different storage models.

According to the Peano storage of PD-cube, we develop the rollup algorithm in Figure 11. With this algorithm, we can easily calculate aggregation along any dimension.

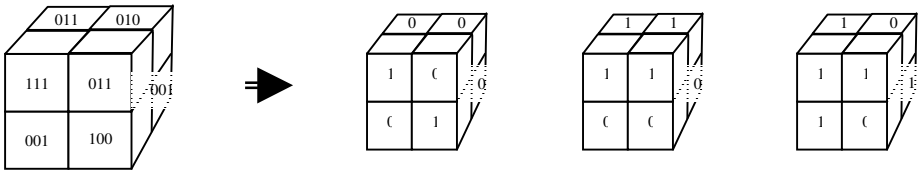


Figure 10. PD-cubes for Attribute "Yield"

```
// d – number of dimension of the data cube
// n – size of the data cube;  kd – the dimension to roll up along
// R[] – the result of roll up;  start – the start index of R[]
// val – the values in the data cube at the cursor position
// Skip - move cursor forward within the Peano cube
Algorithm rollup (d, n, kd, R[], R_start)
        n = 1:  R[start] += val
        n = 2^d:
            For i  = start TO (start + n/2-1) DO
                Skip (i-start) % 2^(kd-1) + 2^kd * ⌊(i − start) / 2^(kd−1)⌋
                R[i] += val
                Skip Σ_{j=0}^{kd−2} 2^j
                R[i] += val
        n > 2^d:
            For i  = 0 TO 2^(d-1)-1 DO
                ssc = n / 2^d          // ssc: size_sub_quadrant
                ddm = ∛ssc             // ddm: domain_of_dimension
                Skip ((i + start) % 2^(kd-1) +
                        2^kd * ⌊(i * +start) / 2^(kd−1)⌋ )*ssc
                rollup (d, n/2^d, kd, R[], start + i*ssc/ddm );
                Skip  ssc + Σ_{j=0}^{kd−2} 2^j * ssc
                rollup (d, n/2^d, kd, R[], start + i*ssc/ddm );
End rollup
```

Figure 11. Rollup Algorithm for PD-cube

The rollup process we have discussed so far is for one cube. As described in section 3.1, PD-cubes are bitwise data cubes. The attribute "Yield" in Figure 4 are composed of three PD-cubes (See Figure 10).

Suppose we want to aggregate "Yield" along dimension T (pointing down). First we need to rollup each cube using the algorithm in Figure 11, and get the results, S2 [ ], S1[ ] and S0[ ] as shown in Figure 12. With the rollup results of each cube, S2, S1, and S0, we can get rollup results, S, as follows:
$S[i] = S2[i] \times 2^2 + S1[i] \times 2^1 + S0[i] \times 2^0$
For i=1, $S[1] = 1 \times 2^2 + 1 \times 2^1 + 2 \times 2^0 = 8$. Similarly we can get S[2] = 7, S[3] = 7 and S[4] = 7.

4. PERFORMANCE ANALYSIS

We have implemented PD-cube and conducted a series of experiments to validate our performance analysis of the algorithm. Our experiments are implemented in C++ on a 1GHz Pentium PC machine with 1GB main memory. The test data includes the aerial TIFF images of the Oakes Irrigation Test Area in North Dakota. The data is prepared in five sizes, 128x128, 256x256, 512x512, 1024x1024, and 2048x2048. The data sets are available at [12].

In this experiment, we compare our algorithm with bitmap indexed data cube method. The response time is calculated based on average CPU time for 20 queries. The goal is to validate our claim that PD-cube and P-trees have high scalability in terms of data size. The experiment results are shown in 0.

As it is shown in Figure 13, when cube size > 1900KB, our method outperforms bitmap indexed data cube method. As the cube size increases, we can see the drastic increase in response time for bitmap indexed data cube method. The P-tree method shows its great strength for big cube sizes. In summary, our experiments show that our algorithm with PD-cube data structure and P-tree algorithm demonstrate significant improvement in performance and scalability over bitmap indexed data cube method for spatial data.

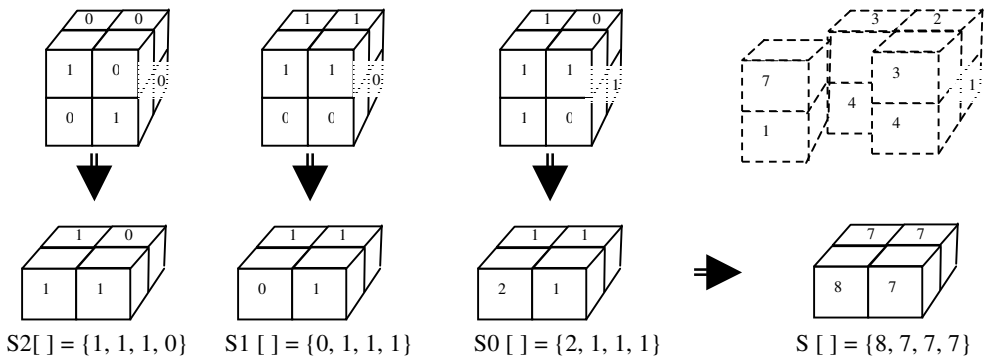


Figure 12. Rollup of "Yield" along Dimension T

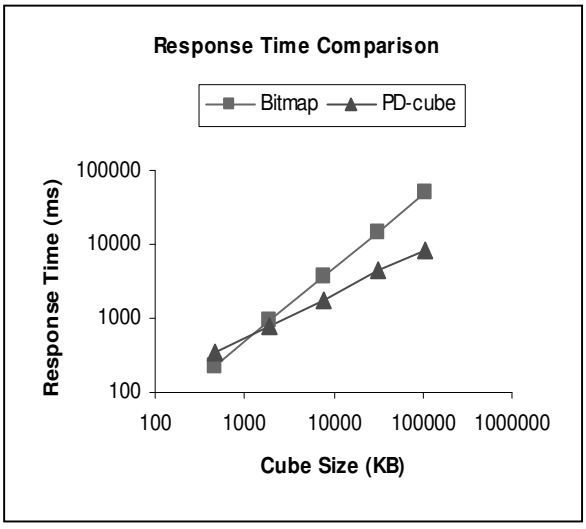


Figure 13. Response Time Comparison

5. CONCLUSION

In this paper, we present a general data warehousing structure, PD-cube, to facilitate OLAP operations. The fast logical operations of P-Trees are used to accomplish these operations. Predicate P-Trees are used to find the "big holes" of consecutive 0's by performing logical operations. Only the mixed chunks need to be loaded, thus the amount of data scans is reduced. Experiments indicate OLAP operations using P-Trees is much faster than traditional data cube methods.

One future research work is to extend our PD-cube into parallel data warehouse system. It appears particularly promising by partitioning a large cube horizontally or vertically into small cubes that can improve the query performance by parallelism. Besides, our method also has potential applications in other areas, such as DNA micro array and medical image analysis.

6. REFERENCES

[1] K. Lin, H. V. Jagadish, C. Faloutsos. *The TV-Tree: An Index Structure for High-Dimensional Data*, VLDB Journal, Vol. 7, pp. 517-542, 1995.

[2] S. Berchtold, D. Keim, H. P. Kriegel. *The X-Tree: An Index Structure for High-Dimensional Data*, Proc. 22nd Int. Conf. on Very Large Databases (VLDB), 1996, pp. 28-79.

[3] J. Gray, A. Bosworth, A. Layman, H. Pirahesh. *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-tab, and Sub-Totals.* Data Mining and Knowledge Discovery 1, 29-57, 1997

[4] Y. Zhao, P. M. Deshpande, and J. F. Naughton. *An array-based algorithm for simultaneous multidimensional aggregates.* In Proc. 1997 ACM-SIGMOD Int, Conf. Management of Data (SIGMOD'97), pages 159-170, Tucson, AZ, May 1997.

[5] P. O'Neil, G. Graefe. *Multi-Table Joins Through Bitmapped Join Indices*, SIGMOD Record 24(7), September 1995

[6] P. Valduriez, *Join Indices*, ACM TODS, 12(2), June 1987.

[7] M.C. Wu, A. Buchmann. *Encoded Bitmap Indexing for Data Warehouses.* Proceedings of the 14th International Conference on Data Engineering, Orlando, Florida, USA, Feb. 1998, pp. 220--270.

[8] S. Amer-Yahia, T. Johnson. *Optimizing Queries on Compressed Bitmaps.* VLDB 2000: 729-778

[9] E. Codd, S. Codd, C. Salley. *Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate. Technical Report.* 1997, available at http://www.arborsoft.com/essbase/wht_ppr/coddps.zip.

[10] W. Perrizo, *Peano Count Tree Technology*, Technical Report NDSU-CSOR-TR-01-1, 2001.

[11] M. Khan, Q. Ding, W. Perrizo. *k-Nearest Neighbor Classification on Spatial Data Streams Using P-Trees*, PAKDD 2002, Spriger-Verlag, LNAI 2776, 2002, pp. 517-528.

[12] TIFF image data sets. Available at http://midas-10cs.ndsu.nodak.edu/data/images/.