

Range CUBE: Efficient Cube Computation by Exploiting Data Correlation*

Ying Feng Divyakant Agrawal Amr El Abbadi Ahmed Metwally

Department of Computer Science
University of California, Santa Barbara

Email: {yingf, agrawal, amr, metwally}@cs.ucsb.edu

Abstract

Data cube computation and representation are prohibitively expensive in terms of time and space. Prior work has focused on either reducing the computation time or condensing the representation of a data cube. In this paper, we introduce Range Cubing as an efficient way to compute and compress the data cube without any loss of precision. A new data structure, range trie, is used to compress and identify correlation in attribute values, and compress the input dataset to effectively reduce the computational cost. The range cubing algorithm generates a compressed cube, called range cube, which partitions all cells into disjoint ranges. Each range represents a subset of cells with the same aggregation value, as a tuple which has the same number of dimensions as the input data tuples. The range cube preserves the roll-up/drill-down semantics of a data cube. Compared to H-Cubing, experiments on real dataset show a running time of less than one thirtieth, still generating a range cube of less than one ninth of the space of the full cube, when both algorithms run in their preferred dimension orders. On synthetic data, range cubing demonstrates much better scalability, as well as higher adaptiveness to both data sparsity and skew.

1 Introduction

Data warehousing is an essential element for decision support technology. Summarized and consolidated data is more important than detailed and individual records for decision maker to make better and faster decisions. Data warehouses tend to be orders of magnitude larger than operational database in size since they contain consolidated and historical data. Most workloads are complex queries which access millions of records to perform several scans, joins and aggregates, and the query response times are more

critical for online data analysis applications. A warehouse server typically presents multidimensional views of data to a variety of analysis and data mining applications to facilitate complex analyses. These applications usually require grouping by different sets of attributes. The data cube was proposed [7] to precompute the aggregation for all possible combination of dimensions to answer analytical queries efficiently. For example, consider a sales data warehouse: (Store, Date, City, Product, Price). Attributes Store, Date, City and Product are called *dimensions*, which can be used as group-by attributes. Price is a numeric *measure*. The dimensions together uniquely determine the measure, which can be viewed as a value in the multidimensional space of dimensions. A data Cube provides aggregation of measures for all possible combinations of dimensions. A *cuboid* specifies a subset of dimensions so that all tuples, with the same values on these dimensions, are aggregated. Each cuboid comprises a set of *cells*, which summarize over tuples with specific values on the group-by dimensions. A cell $a = (a_1, a_2, \dots, a_n, measures_a)$ is an m -dimensional cell if exactly m values among a_1, a_2, \dots, a_n are not *. Here a_i is an attribute value on dimension A_i . For example, consider the base table given below.

Example 1

Store	City	Product	Date	Price
Gateway	NY	DESKTOP	12/01/2000	\$ 100
CompUSA	NY	DESKTOP	12/02/2000	\$ 100
CompUSA	SF	LAPTOP	12/03/2000	\$ 150
Staple	LA	LAPTOP	12/03/2000	\$ 60

The cuboid (Store, *, *, *) contains three 1-dimensional cells: (Gateway, *, *, *), (CompUSA, *, *, *), and (Staple, *, *, *). (CompUSA, NY, *, *) and (CompUSA, SF, *, *) are 2-dimensional cells in cuboid (Store, City, *, *).

The data cube computation can be formulated as a process that takes a set of tuples as input, computes with or without some auxiliary data structure, and materializes the aggregated results for all cells in all cuboids. A naive representation of the cube is to enumerate all cells. Its size is usu-

*This work was supported in part by NSF under grants EIA 00-80134 and EIA 98-17432 and NSF 02-23022.

ally much larger than that of input dataset, since a table with n dimension results in 2^n cuboids. Thus, most work is dedicated to reduce either the computation time [9, 4, 17, 12] or the final cube size [13, 10, 15]. These cost reduction are all without loss of any information, while some others like “quasi-cube” [3, 2], iceberg cube [6] reduce the costs by ignoring “trivial” information. The previous lossless approaches can be classified into three categories (Figure 1) in terms of how the data is compressed before and after the computation.

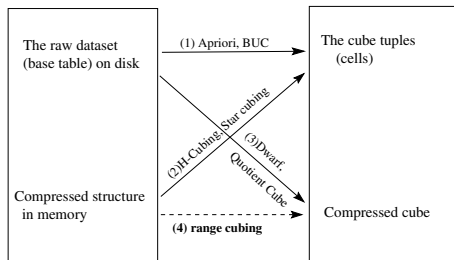


Figure 1. Classification of recent works on cube computation

The Apriori-like algorithm [1] and the Bottom-UP Computation (BUC) [4] approaches are the most influential works among the first category of works for both iceberg cube and full cube computation as shown in Figure 1. These approaches scan a dataset on disk and generate all cells one by one on disk without compressing either the base table or the resulting cube. BUC is especially targeted for sparse datasets, which is usually the case with real data. The performance deteriorates when applied to skewed or dense data.

Han et al. [9] devised a new way for cube computation, called H-Cubing, by introducing a compressed structure of the input dataset, H-Tree. The dataset is compressed by prefix sharing to be kept in memory. It also generates cells one by one and does well on dense and skewed dataset. H-Cubing, as well as an improved method called star-cubing, are the typical works in the second category shown in Figure 1. Both approaches organize input tuples in a hyper-tree structure, called either H-Tree or star-tree. Each level represents a dimension, and the dimension values of each base tuple are populated on the path of depth d from the root to a leaf. In an H-Tree, nodes with the same value on the same level are linked together with a side link. A head table is associated with each H-Tree to keep track of each distinct value in all dimensions and link to the first node with that value in H-Tree. The data structure is revised in [16] as a star tree, which excludes the side links and head tables in an H-Tree. Also a new traversal algorithm is proposed to take advantage of both simultaneous aggregation and prun-

ing efficiency. Moreover, it prunes the base table using a star table before it is loaded into the star tree to accelerate the computation of iceberg cubes. The efficiency of the new star cubing algorithm lies in its ability to compute the cells in some order of cuboids, so that it can enjoy both the sharing of computation as Array Cube [17] and similar pruning efficiency as BUC. Both can be used to compute both iceberg cubes and full cubes, and were shown to outperform BUC especially when dealing with dense or skewed data in iceberg cube computation.

Another set of work shown in the Figure 1 mainly aims to reduce the cube size. It also scans the input dataset, but generates a compressed cube. Sismanis et al. [14] use the CUBETree or compressed CUBETree, called dwarf, to compress the full cube in a tree-like data structure in memory, by utilizing prefix sharing and suffix coalescing. It requires the memory to accommodate the compressed cube. Some work [15, 10] condenses the cube size by coalescing cells with identical aggregation values. Wang et al. [15] use “single base tuple” compression to generate a “condensed cube”. Lakshmanan et al. [10] generalizes the idea to Quotient Cube, which groups all cells with the same aggregation values into classes. Each class contains all cells with the identical aggregation values. Both of them extend the BUC algorithm to find “single tuple cells” or “classes” during the bottom-up computation of a full cube.

However, these works are all not designed to reduce both computational time and output I/O time to achieve the reduction in overall speed-up. As observed in most efficient cube computation works [16, 4, 12], the output I/O time dominates the cost of computation in high dimension and high cardinality datasets since the size of the resulting cube is extremely large, while most compression algorithms focus on how to reduce the final cube size.

In this paper, we propose range cubing, an efficient cube computation method, to generate a compressed data cube, which is both semantics and format-preserving. The motivation of the approach comes from the observation that real world datasets tend to be correlated, that is, dimension values are usually dependent on each other. For example, Store “Starbucks” always makes Product “Coffee”. In the weather dataset used in our experiments, the Station Id will determine the value of Longitude and Latitude. The effect of correlated data on the corresponding data cube is to generate a large number of cells with the same aggregation values. The basis of our approach is to capture the correlation or dependency among dimension values by using a range trie, and generating a range representing a sequence of cells each time.

The range cubing cuts down both computational time and output I/O time without loss of precision. The efficiency of the cube computation comes from three aspects: (1) It compresses the base table into a range trie, a

compressed trie, so that it calculates cells with identical aggregation values only once. (2) The traversal takes advantage of simultaneous aggregation, that is, the $m - dimensional$ cell will be computed from a bunch of $(m + 1) - dimensional$ cells after the initial range trie is built. At the same time, it facilitates Apriori pruning. (3) The reduced cube size requires less output I/O time. Thus, we reduce the time cost not only by utilizing a data structure to efficiently compute the aggregation values; but also by reducing the disk I/O time to output the cube.

Proposed as the fourth category in Figure 1, range cubing integrates the benefits of the two previous categories of algorithms to achieve both computational efficiency and space compression. It compresses a base table without loss of information into a compressed trie data structure called range trie, and computes and generates the subsets of cells with identical aggregation values together as a range. Furthermore, the cube computation algorithm enjoys both the sharing of computation and pruning efficiency of iceberg cubes.

The rest of the paper is organized as follows: Section 2 provides the theoretical basis of our work. Section 3 describes the *range trie* data structure and construction procedure, followed by Section 4 where the range cube is presented. Section 5 develops the range cubing algorithm. Section 6 analyzes the experimental results to illustrate the benefits of the proposed method. We conclude the work in Section 7.

2 Background

We provide some notions needed later in this section. A *cuboid* is a multi-dimensional summarization of a subset of dimensions and contains a set of cells. A data cube can be viewed as a lattice of cuboids, which contains a set of cells.

Now, we define the “ \preceq ” relation on the set of cells of a data cube.

Definition 1 If $a = (a_1, a_2, \dots, a_n)$ is a k -dimensional cell and $b = (b_1, b_2, \dots, b_n)$ is an l -dimensional cell in an n -dimensional cube, define the relation “ \preceq ” between a and b :

$$a \preceq b \text{ iff } k \geq l \text{ and } a_i = b_i \forall b_i \neq * \text{ and } 1 \leq i \leq n.$$

If S_{cells} is the set of cells in a cube, (S_{cells}, \preceq) is a partially ordered set as shown in the next lemma.

Lemma 1 The relation “ \preceq ” on the set of cells in a cube is a *partial order*, and the set of cells together with the relation is a *partially ordered set*.¹

¹Some papers [10] regard it as a lattice of cells. This is the case only if we add a virtual empty node \preceq all base nodes

The proof is to show the relation “ \preceq ” on the set of cells is reflexive, antisymmetric, and transitive.

Intuitively, cells $a \preceq b$ means that a can roll-up to b and the set of tuples to compute the cell a is a subset of that to compute the cell b .

Example 2 Consider the base table in Figure 2(a). Figure 2(b) is the lattice of cuboids in the data CUBE of our example.

From the definition of the containment relation “ \preceq ”, $(S1, C1, *, *) \preceq (S1, *, *, *)$. Also $(S1, *, P1, *) \preceq (S1, C1, P1, *) \preceq (S1, C1, P1, D1)$.

Then we introduce the notion of *range* which is based on the relation “ \preceq ”.

Definition 2 Given two cells c_1 and c_2 , $[c_1, c_2]$ is a *range* if $c_1 \preceq c_2$. It represents all cells, c , such that $c_1 \preceq c \preceq c_2$. we denote this as $c \in [c_1, c_2]$.

Then in the above example, the range $[(S1, *, P1, *), (S1, C1, P1, D1)]$ contains four cells: $(S1, *, P1, *)$, $(S1, C1, P1, *)$, $(S1, *, P1, D1)$, and $(S1, C1, P1, D1)$. Moreover, all four cells in the range have identical aggregation value. So it means that we can use a range to represent these four cells without loss of information.

Lakshmanan et al. [10] developed a theorem to prove the semantic-preserving property of a partition of a data cube based on the notion of *convex partition*.

Definition 3 The *Convex Partition* [5] over a partial ordered set: each partition contains the whole range $[a, b]$ if and only if it contains a and b , and $a \preceq b$.

If we define a partition based on “range”, then it must be convex. As we will show later, a range cube is such a convex partition consisting of a collection of “ranges”.

3 The Range Trie

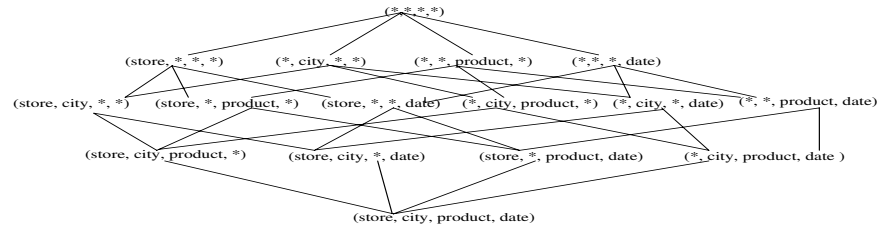
3.1 Basic Definitions

A range trie is a compressed trie structure constructed from the base table. Two types of information are stored in a range trie: dimension values and measure values. The dimension values of tuples determine the structure of a range trie, and are stored in the nodes along paths from the root to leaves. The measure values of tuples determine the aggregation values of the nodes.

Each leaf node in a range trie, like in a star tree [16] or an H-Tree [9], represents a distinct tuple. Thus, a traversal from the root to a leaf node reveals all the information for any tuple(s) with a distinct set of dimension values represented by this leaf node. Each node in a range trie may have

Store	City	Product	Date	Price
S1	C1	P1	D1	\$ 100
S1	C1	P2	D2	\$ 50
S2	C2	P1	D2	\$ 200
S2	C1	P1	D2	\$ 120
S2	C3	P2	D2	\$ 40
S3	C3	P3	D1	\$ 250

(a) the base table



(b) the lattice of cuboids derived from the base table in (a)

Figure 2. The example for cells and cuboids for a base table

several shared dimension values as a **key**, which is different from H-Trees [9] or star trees [16] since the key in these trees is one dimension value per node.

In a range trie, the subset of dimension values stored in a node is shared by all tuples corresponding to the leaf nodes below the node. Therefore, each node represents a distinct group of tuples. Consequently, its aggregation value is that of these tuples. Also, each node can be regarded as the root of a sub-trie defined on a sequence of dimensions appearing in all its descendants. That is, node (S1, C1) is the root of a sub-trie defined on dimensions (Product, Date) of two tuples.

Consider a range trie on an n-dimensional dataset, ordered as A_1, A_2, \dots, A_n . The **start dimension of a range trie** is defined as the dimension with the smallest index, A_1 . Similarly, the smallest of all dimensions appearing in node i and its descendants is chosen as **the start dimension of node i** . So, if the key of a node i in the range trie is the values on a subset of the dimensions, $(A_{i1}, A_{i2}, \dots, A_{ik})$, where $i1 < i2 < \dots < ik$, then the start dimension of node i is the smallest one, A_{i1} . In the example shown in Figure 3(c), the start dimension of both the range trie and the node (S1, C1) is Store.

We have the following relationship between the start dimension of a range trie and that of a node. We assume the key of the root of a range trie is empty. If it is not empty, we exclude those dimensions in the root from the dimensions of the range trie, since the dimension values in the root are common to all tuples.

Proposition 1 *Sibling nodes have the same start dimension. The start dimension of a range trie is the start dimension of its first level child node.*

Figure 3(c) shows a range trie constructed from the table in Example 2. For simplicity, we omit the aggregation values for each node. The number in each node is the number of tuples stored below the node. The dimension order is Store, City, Product, Date. Two data tu-

ples are stored below the node (S1, C1, \$150): (P1, D1, \$100) and (P2, D2, \$50). The start dimension of the node (P1, D1) is Product, which is the smallest dimension among all the dimensions left. These two child nodes of the sub-trie rooted at (S1, C1) have the same start dimension Product, but distinct values P1 and P2. Compared to the star-tree or H-Tree (Fig 3(d)) constructed from the example dataset, the range trie is more compact with more sharing of dimension values

We now present a formal definition of a range trie.

Definition 4 *A range trie on dimensions $A_{i1}, A_{i2}, \dots, A_{in}$ is a tree constructed from a set of data tuples. If the set is empty, the range trie is empty. If it contains only one tuple, it is a tree with only one leaf, whose key is all its dimension values. If it has more than one tuple, it satisfies the following properties:*

1. *If the key of the root node is (a_{j1}, \dots, a_{jr}) , all the data tuples stored in the range trie have the same dimension values (a_{j1}, \dots, a_{jr}) .*
2. *Children have distinct values on their start dimension.*
3. *Each child of the root node is the root of a range trie on the subset of dimensions $\{A_{i1}, \dots, A_{in}\} - \{A_{j1}, \dots, A_{jr}\}$, which preserves the ordering of $\{A_{i1}, \dots, A_{in}\}$.*

We can infer the following properties of a range trie from its definition:

1. The maximum depth of the range trie is the number of dimensions n .
2. The number of leaf nodes in a range trie is the number of tuples with distinct dimension values, which is bounded by the total number of tuples.
3. Because siblings have distinct values on their start dimension, the fan-out of a parent node is bounded by the cardinality of the start dimension of its child nodes.

4. Each interior node has at least two child nodes, since its parent node contains all dimension values common to all child nodes.
5. Each node represents a set of tuples represented by the leaf nodes below it, and we will show later that the set of tuples is used to compute a sequence of cells.

A range trie captures data correlation by finding dimension values that imply other dimension values. We now formalize the notion of data correlation in the following.

Definition 5 *If tuples in the dataset with the values a_k, \dots, a_l on dimensions A_k, \dots, A_l always have values a_s, \dots, a_t on dimensions A_s, \dots, A_t , then a_k, \dots, a_l imply a_s, \dots, a_t , denoted as $(a_k, \dots, a_l) \rightarrow \{a_s, \dots, a_t\}$.*

It is clear to see that if a_k, \dots, a_l imply a_s, \dots, a_t , then any superset of a_k, \dots, a_l will imply a_s, \dots, a_t . According to this notation, we have $(S1) \rightarrow \{C1\}$ and $(S1, P1) \rightarrow \{C1, D1\}$ in the example.

Lemma 2 *Given a node with the key $(a_{i1}, a_{i2}, \dots, a_{ir})$, where a_{i1} is the start dimension value of the node, this start dimension value together with the start dimension values of the node's ancestors imply the dimension values a_{i2}, \dots, a_{ir} .*

The intuition is that the dimension values in a node are implied by the dimension values of its ancestors and its start dimension value. The dimension values of its ancestors are implied by the ancestors' start dimension values. So all the start dimension values of the node and its ancestors imply the dimension values in the node. In the range trie of our example (Figure 3), $(S1, P1) \rightarrow \{D1\}$ since P1 and S1 are the start dimension values of the node (P1, D1) and its ancestor (S1, C1) separately. Also, $(S1) \rightarrow \{C1\}$.

Thus, the set of dimension values implied by $(S1, P1)$ is $\{C1, D1\}$, since $(S1) \rightarrow \{C1\}$ leads to $(S1, P1) \rightarrow \{C1\}$ also. In general, given a node in a range trie with k dimensions on the node and its ancestors, and with l start dimensions for the node and its ancestors, Lemma 2 states that the set of l start dimensions jointly implies those $k - l$ non-start dimensions.

Lemma 2 shows that a range trie identifies data correlation among different dimension values. Lemma 3 explains that data correlation captured in a range trie reveals cells sharing identical aggregation values.

Lemma 3 *Suppose we have an n -dimensional range trie, given any node with k dimensions existing on the node and its ancestors, that is, with dimension values $a_{i1}, a_{i2}, \dots, a_{ik}$ on the dimensions $A_{i1}, A_{i2}, \dots, A_{ik}$. Among these, l dimensions are the start dimensions of the node and its ancestors, say $A_{i1}, A_{i2}, \dots, A_{il}$, where $l \leq k$. Let $c1$ be a $k - l$ -dimensional cell with dimension values $a_{i1}, a_{i2}, \dots, a_{ik}$*

on the dimensions $A_{i1}, A_{i2}, \dots, A_{ik}$ and $c2$ be an $l - l$ -dimensional cell with dimension values $a_{i1}, a_{i2}, \dots, a_{il}$ on the dimensions $A_{i1}, A_{i2}, \dots, A_{il}$. Any c such that $c1 \preceq c \preceq c2$ will have the same aggregation value as $c1$ and $c2$.

The intuition is straightforward. As stated before, the set of start dimension values $(a_{i1}, a_{i2}, \dots, a_{il})$ imply those non-start dimension values $(a_{i(l+1)}, a_{i(l+2)}, \dots, a_{ik})$. The set of tuples used to compute the aggregation value of cell $c2$ are those having dimension values $(a_{i1}, a_{i2}, \dots, a_{il})$. According to Definition 5, all tuples with dimension values $(a_{i1}, a_{i2}, \dots, a_{il})$ have the dimension values $(a_{i(l+1)}, a_{i(l+2)}, \dots, a_{ik})$ on dimensions $A_{i(l+1)}, A_{i(l+2)}, \dots, A_{ik}$. So, for any cell c , $c1 \preceq c \preceq c2$, the set of tuples to compute the aggregation value is the same as that to compute the aggregation value of $c2$.

In our example in Figure 3(c), the cells between $(S1, *, P1, *)$ and $(S1, C1, P1, D1)$ have identical aggregation values.

3.2 Constructing the Range Trie

Algorithm 1: Range Trie Construction

```

Input: a set of  $n$ -dimension data tuples;
Output: the root node of the new range trie;
begin
1  create a root node whose key = NULL;
2  for each data tuple  $a = (a_1, a_2, \dots, a_n)$  {
3    currentNode = root;
4    while (there exist some dimension values in  $a$ ) {
5       $a_i$  = the value in  $a$  on the start dimension of currentNode;
6      if no child of currentNode has start dimension value  $a_i$ ;
7        create a new node using dimension values in  $a$  as key;
8        insert the new node as a child of currentNode; break;
9      else, {
10     commonKey = the set of values both in  $c$ 's key and  $a$ ;
11     remove from  $a$  the above common key values, commonKey;
12     if node  $c$  has more key values than commonKey
13       diffKey =  $c \rightarrow$  key - commonKey
14       set node  $c$ 's key value to commonKey;
15       if smallest dimension in diffKey > that of  $c$ 's children
16         append diffKey to  $c$ 's child nodes;
17       else
18         create a new node  $c1$  using diffKey as key values;
19         set  $c$ 's children as  $c1$ 's children;
20         create another new node  $c2$  using values left in  $a$ ;
21         set  $c1$  and  $c2$  as children of node  $c$ ; break;
22       end if;
23     end if;
24     set currentNode to node  $c$ ;
25   } end while
26 } end for
27 return root;
end;
```

Figure 4. The range trie construction

This section will describe the range trie construction algorithm. It is done by a single scan over the input dataset. Each tuple will be propagated downward from the root till a leaf node. Unlike in an H-Tree or a star tree, where each dimension value will be propagated on one node in order, each node of a range trie extracts the common dimension values shared by the tuples in the trie. For the set of remaining dimension values, choose the smallest dimension as the start dimension of the trie and insert the tuple in the branch

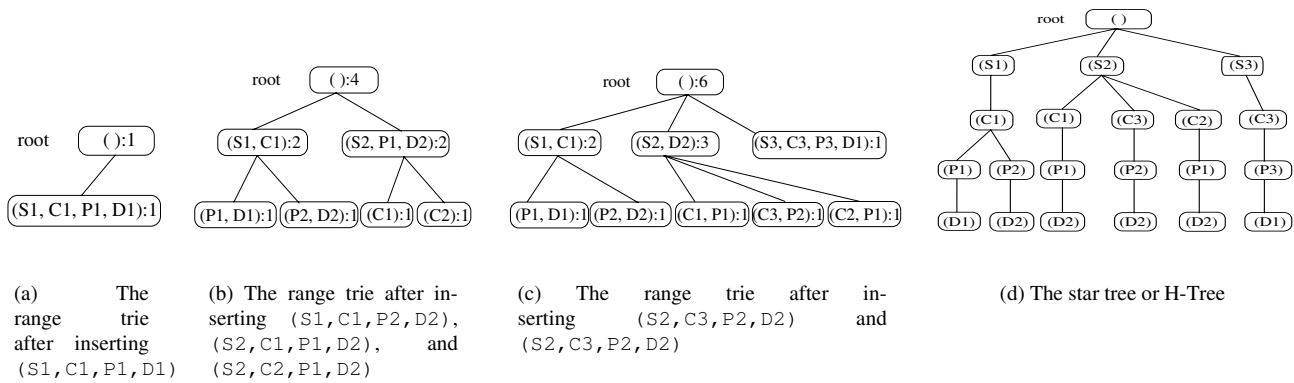


Figure 3. The construction of the range trie

below the node whose start dimension value is equal to the tuple's value on that dimension.

Figure 4 gives the algorithm for constructing a range trie from a given dataset. The algorithm iteratively inserts a tuple into the current range trie. Lines 3-26 describe how to insert dimension values of a tuple downward from the root of the current range trie. First, it searches for the child node whose start dimension value is identical to that of the tuple. If none of the existing child nodes has identical start dimension value as the tuple, the insertion ends by adding a new leaf node as a child of the root node, where all dimension values remaining in the tuple are used as the key of the new node (Line 7-8). For example, when the last tuple $(S3, C3, P3, D1)$ is inserted into the range trie shown in Figure 3(b), a new leaf node is created as the rightmost child node of the root in Figure 3(c).

Otherwise, it chooses the existing child node whose start dimension value is identical to that of the tuple. A set of common dimension values is obtained by comparing the dimension values of the tuple and those in the chosen child node, and is removed from the tuple. This is described in lines 10-11 of the algorithm.

Now if the key values of the chosen child node are all in the set of common dimension values, that is, all the dimension values in the chosen child node appear in the tuple, the root of a range trie is set to the chosen child node. The insertion continues in the next iteration to insert the tuple with fewer dimension values into the range trie rooted at the previously chosen node. If the tuple $(S1, C1, P3, D2)$ is to be inserted in the range trie shown in Figure 3(b), the chosen child node $(S1, C1)$ will not be changed. The dimension values left in the tuple will be $(P3, D2)$ after removing $(S1, C1)$. The new iteration will insert the tuple $(P3, D2)$ to the range trie rooted at the node $(S1, C1)$.

Lines 12-23 describe the situation if some dimension val-

ues in the chosen child node are not in the set of common dimension values. If the smallest dimension in the chosen node excluding the set of common dimensions is larger than the start dimension of the child nodes of the chosen node, append those non-common dimension values to all those child nodes (line 16). Then, the new iteration continues with the tuple with fewer dimensions and the range trie rooted at the chosen node. This is the case when the tuple $(S2, C3, P2, D2)$ is inserted to the range trie in Figure 3(b). The child node $(S2, P1, D2)$ is chosen to insert the tuple. The set of common dimension values is $\{S2, D2\}$ on the dimensions Store, Date. The smallest of the non-common dimensions is Product which is larger than the start dimension City of the child nodes of the chosen node, given the dimension order: Store, City, Product, Date. The set of non-common dimension values $\{P1\}$ is appended to all the children, which become $C1, P1$ and $C2, P1$. The tuple $(C3, P2)$ is inserted in the range trie rooted at the updated chosen node $(S2, D2)$ in the next iteration. This time no child node exists with the identical start dimension value as $C3$. The insertion ends after a new leaf node $(C3, P2)$ is added as a child node of the current root node $(S2, D2)$, as shown in Figure 3(c). If the smallest of the non-common dimensions in the chosen node is smaller than the start dimension of its child nodes, create a new node which uses the set of non-common dimension values as the key and inherit all the child nodes of the chosen node as its children. Then, create another new leaf node which uses the remaining dimension values in the tuple as the key, and set these two new nodes as children of the chosen node. Here, if the chosen node is a leaf node itself, we assume the start dimension of its child nodes is always largest. An example of this case occurs when the tuple $(S1, C1, P2, D2)$ is inserted in the range trie shown in Figure 3(a). The non-common dimension values in the chosen node are $P1, D1$ and thus non-common di-

mensions are Product, Date, where the smallest non-common dimension is Product according to the dimension order given before. It is smaller than the start dimension of the child nodes of the chosen node (S1, C1, P1, D1), since we assume a leaf node has the largest possible dimension in its “child nodes”. The dimension values of the chosen node are replaced by (S1, C1), and a new node is created which has dimension values (P1, D1). Another new leaf node is created with the dimension values left in the tuple, P2 and D2, and both are set as the child nodes of the node (S1, C1). It is shown as the left branch in the Figure 3(b).

The range trie constructed from a dataset is invariant to the order of data entry. The process of constructing a range trie iteratively finds the common dimension values shared by a set of tuples, which is more than prefix sharing as in many other tree-like structure.

The Size of a Range Trie As we will see later, the number of nodes in a range trie is an important indicator of the efficiency of the range cube computation. A range trie with fewer nodes will generally take less memory and less time to compute the cube, and the resulting cube will be more compressed. Suppose we have a range trie on a D -dimensional dataset with T tuples. It can have no more than T leaf nodes. The depth of the range trie is D in the worst case, when the dataset is very dense and the trie structure is like a full tree. It is $\log_N T$ in the average case, for some N , the average fan-out of the nodes. Now we will give a bound on the number of internal nodes. The proof is by induction on the depth of the range trie.

Lemma 4 *The number of interior nodes in a range trie with T leaf nodes is bounded by $T - 1$, while it is $\frac{T-1}{N-1}$ on average. Here, N is the average fan-out of the nodes.*

Compared to an H-tree, whose internal nodes are $T * (D - 1)$ in the worst case, a range trie is more scalable w.r.t the number of dimensions, that is, its efficiency is less impacted by the number of dimensions.

4 The Range Cube

As shown in Lemma 3, the range trie will identify a range of cells with identical aggregation values. The definition of a range coincides with the definition of a sub-lattice.

Two cells c_1 and c_2 are needed to represent a range. However, since $c_1 \preceq c_2$, it means intuitively that either c_1 has the same dimension value as c_2 or c_2 has the value “*” in place of c_1 's value on each dimension. For each dimension value a , we introduce a new symbol \bar{a} to represent either the value a or *. Thus, a range can be represented as a **range tuple** with n values followed by its aggregation value.

Definition 6 *Let $[a, b]$ be a range. If $a = (a_1, a_2, \dots, a_n)$ is a k -dimensional cell and $b = (b_1, b_2, \dots, b_n)$ is an l -dimensional cell, and $a \preceq b$. The **range tuple** $c = (c_1, c_2, \dots, c_n, aggregate_value)$ is defined as: For $i = 1, 2, \dots, n$,*

$$c_i = \begin{cases} a_i & \text{if } a_i = b_i \\ \bar{a}_i & \text{if } a_i \neq * \text{ and } b_i = * \end{cases}$$

*And the **aggregate_value** is the aggregation value shared by all cells in the range.*

A **range cube** is a partitioning of all cells in a cube and each partition is a range. All ranges in a range cube are disjoint and each cell must be in exactly one range.

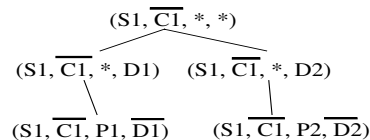


Figure 5. The ranges with Store set to “S1”

A range cube is basically a compressed cube, which replaces a sequence of cells with a range. For example, $(S1, \bar{C1}, P1, \bar{D1})$ represents all the cells in the range $[(S1, *, P1, *), (S1, C1, P1, D1)]$ which are: $(S1, *, P1, *)$, $(S1, C1, P1, *)$, $(S1, *, P1, D1)$, and $(S1, C1, P1, D1)$. So, the resulting cube size is reduced. As an example, the five ranges in Figure 5 consist of 14 cells.

A range cube has some desirable properties compared to other compressed cubes. First, it preserves the native representation of a data cube, i.e., each range is expressed as a tuple with the same number of dimensions as a cell. So, a wide range of current database and data mining applications are compatible with range cubes. In addition, other compression or index techniques such as dwarf [14] can also be applied naturally to a range cube. This property makes it a good candidate to incorporate with other performance improvement approaches.

Second, as we will show, a range cube preserves the semantic properties of a data cube, which means it keeps the roll-up/drill-down property of a data cube as described in [10]. Lakshmanan et al. proposed the semantics-preserving property for a compressed cube in [10]. They developed several theorems to help prove the semantics-preserving property of a compressed cube. We will use them to show the semantics-preserving property of a range cube. First it is clear from the definition that a range cube is a convex partition of a full data cube. Now we reach the conclusion that a range cube is semantics-preserving.

Theorem 1 *A range cube preserves the roll-up/drill-down properties of a data cube.*

This follows from the formalization of partition preserving the roll-up/drill-down semantics of a cube in [10]. Lakshmanan et al. [10] show that (1) a partition is convex *iff* the equivalence relation inducing the partition is a weak congruence; and (2) a weak congruence relation respects the partial order of cells in a cube. So it immediately leads to the conclusion that the range cube is a partition which respects the partial order of cells, that is, it preserves the roll-up/drill-down property of the original cube.

As an example, Figure 5 shows the roll-up and drill-down relation of the ranges on those cells whose Store dimension value is ‘‘S1’’.

5 The Range Cube Computation

The range cubing algorithm operates on the range trie and generates ranges to represent all cells completely. The dimensions of the data cube are ordered, yielding a list A_1, A_2, \dots, A_n . It iteratively reduces an n -dimensional range trie to an $(n-1)$ -dimensional range trie after traversing the n -dimensional range trie to generate ranges and recursively apply the range cubing on each node. The n -dimensional range trie generates the cells in cuboids from $(A_1, *, \dots, *)$ to (A_1, A_2, \dots, A_n) . And then the $(n-1)$ -dimensional range trie generates cells in cuboids from $(*, A_2, \dots, *)$ to $(*, A_2, \dots, A_n)$ and so on.

5.1 An Example

In this section, we will illustrate the efficiency of range cube computation using the previous example shown in Fig 2(a). Fig 3(c) is the initial four-dimensional range trie constructed from the base table. The number inside each node is the number of tuples used to compute the aggregation values of each node. The algorithm visits the node $(S1, C1)$ and produces the range $(S1, \bar{C1}, *, *)$, representing the cells $(S1, *, *, *)$ and $(S1, C1, *, *)$. Then it applies the range cubing algorithm on the sub-trie rooted at node $(S1, C1)$, which is defined on dimensions Product and Date. The range cubing on the sub-trie should generate the ranges containing all cells with dimension values either $(S1, *)$ or $(S1, C1)$, that is, all the cells with dimension values $(S1)$. Similarly, the middle branch of the root will generate ranges representing all the cells with the dimension value $(S2)$ and the right-most branch will generate those having the dimension value $(S3)$.

After traversing the initial range trie in Fig 3(c), it is reorganized to generate a three-dimensional range trie as shown in Fig 6(a). It is defined on dimensions

City, Product, Date. The way to reorganize a n -dimensional range trie to a $(n-1)$ -dimensional range trie will be explained later. Similar traversal is applied to the three-dimensional range trie (Fig 6(a)), and it generates all the ranges including those cells with values $(*, C1)$ or $(*, C2)$ or $(*, C3)$ on dimensions Store and City.

After that, the three-dimensional range trie is reduced to the two-dimensional range trie on dimensions Product, Date as shown in Fig 6(b). The transformation is obtained by reorganizing the three-dimensional range trie. This time it produces all the ranges representing cells that have values $(*, *, P1)$ or $(*, *, P2)$ on dimensions (Store, City, Product). Finally a one-dimensional range trie is obtained from the two-dimensional range trie as shown in Fig 6(c) and gives the aggregation values for the cells $(*, *, *, D1)$ and $(*, *, *, D2)$.

Now we will explain how to transform a four-dimensional range trie to a three-dimensional range trie on (City, Product, Date). By setting the Store dimension of each child nodes of the root to $*$, the trie will look like the one in Figure 6(d). Since the new start dimension is City, the child nodes which do not contain the dimension Store will append their dimension values to their children. That is, the dimension value D2 is appended to its three children, which become $(C1, P1, D2)$, $(C3, P2, D2)$, $(C2, P1, D2)$. These three nodes will be merged with the nodes $(C1)$ and $(C3, P3, D1)$ according to the third property of a range trie. Then, the resultant three-dimensional range trie appears in Figure 6(a)

5.2 The Range Cubing Algorithm

If we generalize the above example, the algorithm can be obtained as in Figure 7. It starts from an initial n -dimensional range trie, and does a depth-first traversal of the nodes. For each node, it will generate a range and recursively apply the cubing algorithm in case it is not a leaf node. Then it will reorganize the range trie to a $(n-1)$ -dimensional range trie and is recursively applied. This is repeated for the $(n-1)$ -dimensional range trie followed by the $(n-2)$ -dimensional range trie, the $(n-3)$ -dimensional range trie and so on. The algorithm ends with a 1-dimensional range trie.

It is easy to see that the number of recursive calls is equal to the number of interior nodes. So the bound on the interior nodes given previously provides a bound on the computational complexity. Moreover, each node which is generated from the merging of several existing nodes has a distinct aggregation value since it is aggregated from a distinct set of tuples. So it can be shown that it needs the least number of aggregation operations in the algorithm, once the initial range trie is constructed. The proof is not shown here due

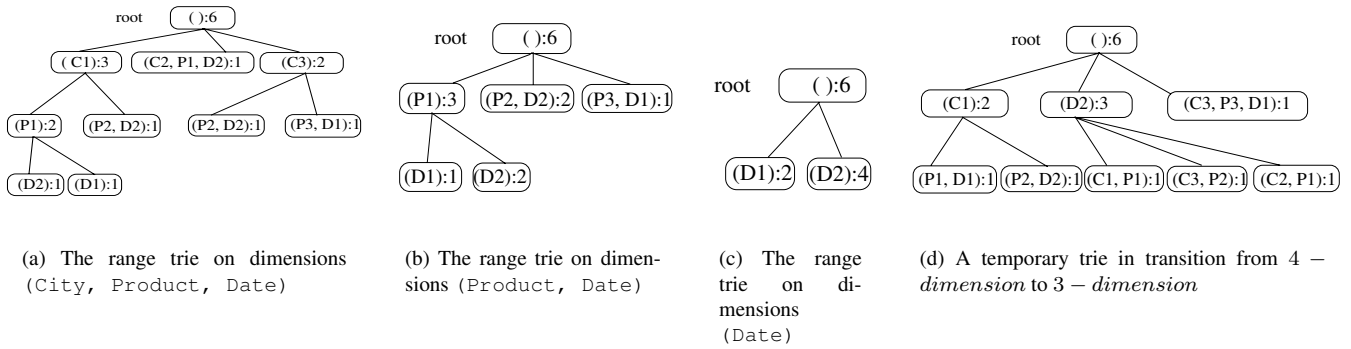


Figure 6. The process of range cube computation

```

Algorithm 2: Range Cube Computation
Input:  root1: the root node of the range trie;
        dim: the set of dimensions of root1;
        upper: the upper bound cell;
        lower: the lower bound cell;
Output: the range tuples;
begin
  for each dimension  $iDim \in dim$  {
    for each child of root, namely,  $c$ , {
      Let startDim be the start Dim of the  $c$ ;
      upper[startDim] =  $c$ 's start dimension value;
      set upper's dims  $\in \{c \rightarrow key - startDim\}$  to *;
      set lower's dims  $\in c \rightarrow key$  to  $c \rightarrow KeyValue$ ;
      outputRange(upper, lower,  $c \rightarrow quant-info$ );
      if  $c$  is not a leaf node
        call range cubing on  $c$  recursively;
    }
    Traverse all the child nodes of the root and
    merge the nodes with same next startDim value;
    upper[iDim] = lower[iDim] = *;
    iDim = next StartDim;
  }
end;

```

Figure 7. Range Cube Computation algorithm

to the space constraint. The intuition is that each node represents a distinct aggregation values and each node representing some n -dimensional cells is aggregated from those nodes representing $(n + 1)$ -dimensional cells. In our example, the total number of aggregation needed is 9 in range cubing.

All existing cube computation algorithms are quite sensitive to dimension order since it determines the order for computing cells. The range trie also depends on the dimension order as shown before. However, it is only used to enforce the choice of start dimensions, that is, the start dimension of a child node must be larger than the start dimensions of its ancestors. The range trie provides some flexibility to dimension order since it not only allows the dimensions in a child nodes to be smaller than the non-start dimensions of the parents, but also allows different branches to have different dimension orders based on the data distribution. This feature makes range cube computation less sensitive to dimension order.

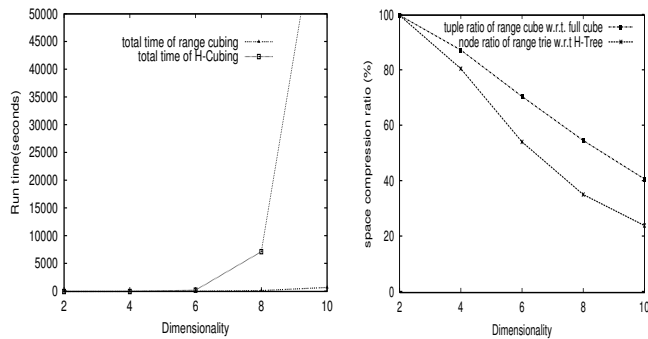
The favorite dimension order for the range cubing is decreasing cardinality, which is the same as star-cubing and BUC. Since dimensions with large cardinality are more likely to imply the dimensions with small cardinality, more dimension values are implied by a value of a large cardinality dimension. Ordering by decreasing cardinality results in less nodes in the range trie. Moreover, this produces more partitions, which means smaller sizes for each partition on the average, and thus achieves earlier pruning, while also generating a more compressed range cube.

Another advantage of the range cubing computation algorithms is that it efficiently identifies cells with the same aggregation values without any sorting operation. Although others have noticed that most cells share the same aggregation values, previous work [14, 11] needs sorting either on the base table or on classes generated temporarily during the execution.

6 Experimental results

To evaluate the efficiency of the range cubing algorithm in terms of time and space, we conducted a comprehensive set of experiments and present the results in this section. Currently there exist two main cube computation algorithms: BUC [4] and H-Cubing [9]. Since in [9] it was shown that H-Cubing outperforms BUC in cube computation, we will compare range cubing with H-Cubing only. As will be shown, the experiments demonstrate that range cubing saves computation time substantially, and reduces the cube size simultaneously. The range cube does not try to compress the cube optimally like Quotient-Cube which finds all cells with identical aggregation values, however, it still compresses the cube close to optimality as shown in the case of real dataset. It balances the computational efficiency and space saving to achieve overall improvements.

Both synthetic and real dataset are used in the experiments. We used uniform and Zipf [18] distributions for



(a) comparison of total run time (b) space compression of range cube

Figure 8. Evaluating the effectiveness of range cubing

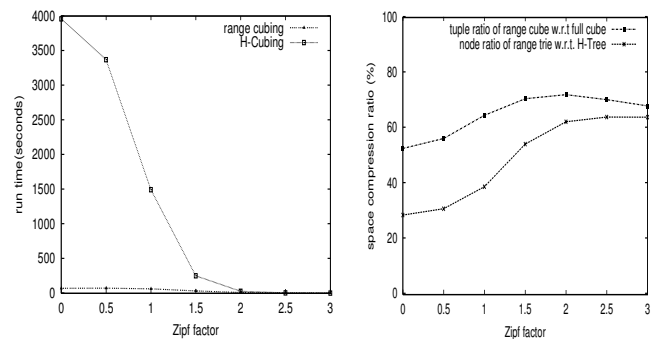
generating the synthetic data. The real dataset is the data of weather conditions at various weather land stations for September 1985 [8]. These are datasets most frequently used to test the performance of cube algorithms [10, 15, 4]. We ran the experiments on an AthlonXP 1800+ 1533MHz PC with 1.0G RAM and 50G byte hard disk.

We measure the effectiveness of range cubing on the basis of three metrics. The first metric is the total run time to generate the range cube from an input dataset, which measures the computational cost. The results about computation time without I/O are omitted since its improvement is no worse than the total run time. The second metric is tuple ratio [15], which is the ratio between the number of tuples in the range cube and the number of tuples in the full cube. Tuple ratio measures the space compression of the resulting cube, and the smaller the ratio the better. The third metric is the *node ratio*, which is defined as the ratio between the number of nodes in the initial range trie and that in the H-Tree. The number of nodes is an important indicator of the memory requirement of the cube computation. For a specific dimension order, the less the number of nodes, the better the performance.

6.1 Synthetic Dataset

First, we demonstrate the effectiveness of range cubing and then show the performance in case of skewness and sparsity. Finally we will investigate its scalability.

Effectiveness. This set of experiments shows the effectiveness of range cubing in reducing time and space costs. We use a synthetic dataset with Zipf distribution. The Zipf factor is fixed at 1.5. We also fix the number of tuples at 200K and the cardinality of each dimension is 100. The



(a) comparison of total run time (b) space compression of range cube

Figure 9. Evaluating the impact of skewness

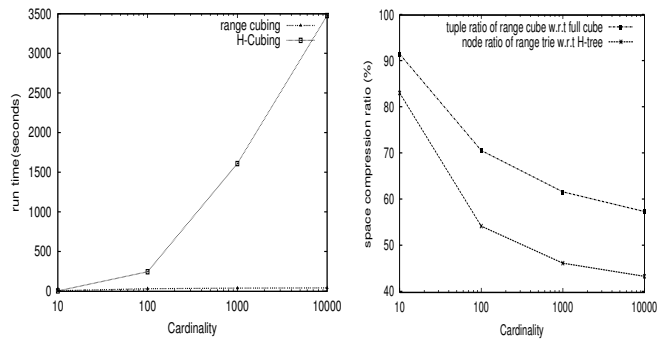
number of dimensions is varied from 2 to 10. Fig 8(a) shows the timing result and Fig 8(b) shows the space compression result.

When the dimensionality increases, both range cubing and H-Cubing need more time and space. However, the possibility of data correlation increases too. So range cubing does not grow as rapidly as H-Cubing does, and achieves better time and space compression ratios. The computation time of range cubing is one eighth of H-Cubing even with a 6-dimension dataset. The space ratio improves as the dimensionality grows.

When the cube is very dense (2 to 4 dimensions), the computation time and cube size of range cubing and H-Cubing are almost the same. This shows that the lower bound of a range trie is an H-Tree and a range cube in the worst case is an uncompressed full cube.

Skewness. We investigate the impact of the data skew in the second set of experiments. We set the number of dimensions to 6 and the cardinality of each dimension to 100 and the number of tuples to 200K. We generate the dataset with the Zipf factor ranging from 0.0 (uniform) to 3.0 (highly skewed) with 0.5 interval. Fig 9(a) and Fig 9(b) show how the computation time and the cube size compression change with regard to increased data skew.

As data gets more skewed, both cubing algorithms decrease in their computation time, because their data structures adapt to the data distribution. The space compression ratio increases with Zipf factor and it becomes stable after Zipf factor reaches 1.5. As data gets skewed, the space compression in a smaller dense region decreases while the space compression in the sparse region increases. When the Zipf factor is about 1.5, the two effects balance. As noted for BUC, and BUC-dup algorithms [4], they perform worse as Zipf factor increases, and perform worst when Zipf factor is 1.5.



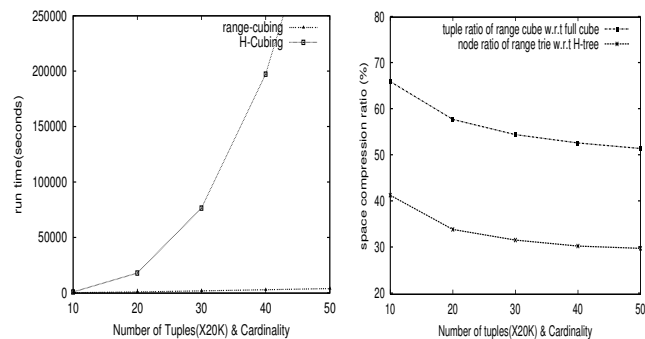
(a) comparison of total run time (b) space compression of range cube

Figure 10. Evaluating the impact of sparsity

Sparsity. To evaluate the effect of sparse data, we vary the cardinality of dimensions while fixing other parameters. This is different from most prior experimental evaluations [10, 15], which vary the number of tuples while fixing the cardinality. The reason is that the latter approach changes both the sparsity of the dataset and the experimental scale (the size of dataset). When we measure the cube space compression ratio, we can assume that when we enlarge a dataset with a specific level of sparsity, the space compression ratio will remain stable. However, a similar assumption cannot be made for the run time. Hence, in order to isolate only the sparsity effect, we vary the cardinality instead of the number of tuples. The results in Fig 10 are based on the dataset with Zipf factor of 1.5, the number of dimensions is 6, and the number of tuples is 200K. The cardinality of the dimensions takes the values 10, 100, 1000 and 10000.

When the cardinality increases, the run time used by the H-Cubing algorithm increases rapidly while that of range cubing does not change much. The space compression ratio improves. The reason is that more data *coincidence* happens when the data is sparse, resulting in a more compressed range trie. This means that on average, a range tuple represents more cells. Meanwhile, the efficiency of H-cubing comes from prefix sharing in the H-tree structure. It has less prefix sharing as the cardinality increases, which makes its performance worse with larger cardinality.

Scalability. We also measured how the algorithm scales up as the dataset size grows. We vary both the number of tuples and the cardinality, in order to accurately measure the effect of experiment scale on the cubing algorithm. Most prior research changed only the number of tuples to measure scalability. However, this will cause the dataset to get denser simultaneously. Then the impact on the algorithm is a hybrid of the increased density and the experiment size. Therefore, we change both so that the dataset density re-



(a) comparison of total run time (b) space compression of range cube

Figure 11. Evaluating the scalability of algorithms

mains stable as the dataset size increases. In our experiments, we use a dataset with 10 dimensions and Zipf factor 1.5. The cardinality ranges from 10 to 50 with 10 interval. The number of tuples varies from 200K to 1M with an interval of 200K.

The results in Fig 11 shows that range cubing is much more scalable. The total run time in case of 200K tuples with cardinality 10 for each of 10 dimensions is 412.3 seconds for range-cubing and 1072.5 for H-Cubing. While H-cubing goes up rapidly and reaches more than 387803 seconds in case of 1M tuples with cardinality 50; Range cubing in this case takes only 4104 seconds. As we can see, the space compression ration is slightly better when the experiment scale goes up, since the data density does not change.

6.2 Real Dataset

We used the weather dataset which contains 1,015,367 tuples. The attributes with cardinalities are as follows: station-id (7,037), longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8) and brightness (2).

The dimension order of the dataset does affect the performance in the range cubing algorithm. As mentioned before, it prefers to order dimensions by decreasing cardinality. The reason is that it allows flexible dimension orders and may have different dimension order on each branch.

As we can see from the results in Fig 12, the range cubing algorithm takes significantly less time and space than H-Cubing. When we order the dimensions by decreasing cardinality, as favored by range cubing, it is more than 30 times faster than H-Cubing in its favorable dimension order (by increasing cardinality) and generates a range cube with only

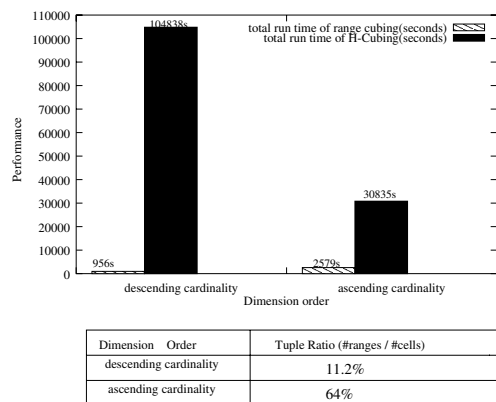


Figure 12. Evaluating performance on the real dataset

11.2% of the full cube size. Even in the least favorable dimension order (increasing cardinality), range cubing takes less than one fortieth of the run time used by H-Cubing in its least favorable order. The space compression ratio of the range cube grows to 64% in its least favorable dimension order.

7 Discussion and Future Work

This paper proposes a new cube computation algorithm, which utilizes the correlation in the datasets to effectively reduce the computational cost. Although it does not compress the cube optimally, its goal is to achieve overall speed-up by balancing computational efficiency and space efficiency. It not only substantially reduces the computational time of cube construction, but also generates a compressed form of the cube, called the range cube. The range cube is a partition of the full cube and effectively preserves its roll-up/drill-down semantics. Moreover, a range is expressed as a tuple similar to a cell, so that it preserves the format of the original dataset. These desirable properties facilitates integration of the range cube with existing database applications and data mining techniques. Since the range trie reduces the effective dimensions of input datasets, it is appropriate for handling high-dimensional sparse or correlated datasets. Even when the dataset is dense, the range trie degrades to the H-Tree, and the range cube is a full cube.

Our current efforts include incorporating constraints with range cube computation, dealing with holistic functions, and applying some other compression techniques to compress the cube. Supporting incremental and batch updates, are to be explored in the future.

References

- [1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 487–499, 1994.
- [2] D. Barbara and M. Sullivan. Quasi-cubes: Exploiting approximation in multidimensional databases. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 12–17, 1997.
- [3] D. Barbara and X. Wu. Using loglinear models to compress data cubes. In *Proc. Int. Conf. on Web Age Information Systems (WAIM 2000)*, pages 311–322, 2000.
- [4] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 359–370, 1999.
- [5] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 1967.
- [6] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 299–310, 1998.
- [7] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 152–159, 1996.
- [8] C. Hahn. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. cdiac.est.ornl.gov/ftp/ndp026b/SEP85L.Z, 1994.
- [9] Jiawei Han, Jian Pei, Guozhu dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 1–12, 2001.
- [10] L. V. S. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. In *Proc. Int. Conf. on Very Large Database (VLDB)*, pages 766–777, 2002.
- [11] Laks V. S. Lakshmanan, P. Jian, and Y. Zhao. Qc-trees: An efficient summary structure for semantic olap. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 64–75, 2003.
- [12] K. A. Ross and D. Srivastana. Fast computation of sparse datacubes. In *Proc. Int. Conf. on Very Large Database (VLDB)*, pages 116–125, 1997.
- [13] J. Shanmugasundaram, U. Fayyad, and PS Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In *Knowledge Discovery and Data Mining*, pages 223–232, 1999.
- [14] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the petacube. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 464–475, 2002.
- [15] W. Wang, J. Feng, H. Lu, and J. Yu. Condensed cube: An effective approach to reducing data cube size. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 155–165, 2002.
- [16] D. Xin, J. Han, X. Li, and B. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 476–487, 2003.
- [17] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 150–170, 1997.
- [18] G.K. Zipf. *Human Behavior and The Principle of Least Effort*. Addison-Wesley, 1949.