Hierarchical Compact Cube for Range-Max Queries

Sin Yeung Lee Tok Wang Ling HuaGang Li

School of Computing
National University of Singapore
{jlee,lingtw,lihuagan}@comp.nus.edu.sg

Abstract

A range-max query finds the maximum value over all selected cells of an on-line analytical processing (OLAP) data cube where the selection is specified by ranges of contiguous values for each dimension. One of the approaches to process such queries is to pre-compute a prefix cube (PC), which is a cube of the same dimensionality and size as the original data cube, but with some pre-computed results stored in each cell.

In this paper, we propose a new cube representation called Hierarchical Compact Cube, which is an hierarchical structure that stores not only the maximum value of all the children sub-cubes, but also stores one of the locations of the maximum values among the children sub-cubes. The storage requirement is much less than the prefix cube methods. Furthermore, both of our analysis and experiment results show that the average query time using our method is bounded by a constant independent on the number of data in the data cube, N. For a fixed dimension, the average update cost of our new structure in the worst case is also relatively low. It is only $O(\log N)$.

1 Introduction

Aggregation is a common and computation-intensive operation in on-line analytical processing systems (OLAP) [3, 4, 7], where the data is usually modelled as a multidimensional data cube [5, 6, 10], and queries typically involve aggregations across various cube dimensions. Formally, an n-dimensional data cube is derived from a projection of $n+1$ attributes from some relation R, where one of these attributes is classified as a measure attribute and the remaining n attributes are used as **dimensional attributes**. Each dimension of the data cube corresponds to a dimensional attribute, and the value in each cube cell is an aggregation of the measure attribute value of all records in R having the same dimensional attribute values. For instance, consider the database which stores the sales of each item in each day for each outlets, the data can be stored in a cube having three dimension --- item, date and outlets. The value in each cube will be the actual sales.

Using the data cube model, we can answer many OLAP range queries [11] efficiently. In particular, we propose a new pre-computation technique for a class of OLAP queries called **range-max queries**. A range-max query finds the maximum value over all selected cells of an OLAP data cube where the selection is specified by a range of contiguous values for each dimension [6]. For example, finding the maximum sales of stationary items (each has an item code ranging from 1200 to 1258) between day 130 and day 136 in all the western outlets (branch-no ranging from 45 to 89) is a range-max query. It can be realized using the following SQL statement:

```
SELECT MAX(amount) FROM sales WHERE
    ( (item>=1200) AND (item <= 1258) ) AND
    ( (day>=130) AND (day<=136) ) AND
    ( (branch>=45) AND (branch<=89) );
```

The most direct approach is a *naïve* approach. We evaluate a range-max query by accessing each individual cell from the data cube itself and find the maximum value. However, the cost of access is proportional to the size of the sub-cube specified by the range. To illustrate, given a 10-dimension data cube, if we double the size of each dimension in the range, we will increase the total access cost by 1024 times. This is clearly unacceptable.

Proceedings of the 26th Internal Conference on Very Large Databases, Cario, Egypt, 2000.

Note that this naïve method can be applied to other aggregate functions such as SUM.

To improve the range query for the aggregate function SUM, considerable research has been done in the database community [8, 9, 11, 12, 13]. One of the foundation stones for efficient range-sum query algorithm is to pre-compute a set of summary results [9] which will be used to speed up the processing of an OLAP query of arbitrary range. The most commonly found ideas is the Prefix Sum Method. In this method, a prefix cube *PC*, of the same size as the data cube *DC*, stores various pre-computed prefix aggregation. In particular, $PC\langle x_1,\ldots,x_d\rangle$ stores the sum of all the data in *DC* ranging from <0,…,0> to $\langle x_1,\ldots,x_d\rangle$. With the use of *PC*, any range-sum query on d dimension can be answered with a constant (2^d) cell accesses. To illustrate, the sum of all the data in *DC* ranging from <2,4> to <6,9> can be computed with only four cell accesses of the *PC* by using the formula:

sum(<2,4>, <6,9>) =
sum(<0,0>, <6,9>) – sum(<0,0>, <6,3>) –
sum(<0,0>, <1,9>) + sum(<0,0>, <1,3>)

or alternatively,

sum(<2,4>, <6,9>) = *PC*<6,9> – *PC*<6,3> –
PC<1,9> + *PC*<1,3>

Although the Prefix Sum Method has a very good constant time query cost, it is very expensive to update the prefix sum cube. A single update on the data at *DC*<0,…,0> requires to update every cell in the *PC*. Other methods try to correct this weakness. For example, the Relative Prefix Sum method [12] has a constant query cost and a much reduced $O(n^{d/2})$ update cost. This achieves a better overall effect for frequently updated data cube. The Hierarchical Cubes method [13] further improves [12] to allow a dynamic fine-tuning between the query cost and update cost.

Despite all these works on range-sum query, they cannot be directly applied to the range-max query. In particular, most of the existing range-SUM methods explore the idea that, given two disjointed regions A and B,

$$sum(B) = sum(A+B) - sum(A)$$

where $A + B$ is the union of the two regions. This equality is exactly the corner stone to make prefix sum works. However, for the case of range-max query, even if we know the maximum value of both regions A and $A+B$, we still cannot decide the maximum value of the region B.

Fortunately, there are many other aspects that we can explore to speed up the range-max query that the range-sum query does not process:

1. In a range-sum query, it is possible to prune some processes in the search for the maximum. In particular, given three regions A, B and C. If it is known that max(A+B) is not more than max(C), then both max(A) and max(B) are smaller than max(C). Therefore, we do not need to explore regions A nor B to find the exact value of max(A) nor max(B). Generalising this idea, if a requested range is covered by regions $A_1,\ldots,A_n$, we can prune off any further investigation on A_i if the maximum value of A_i is not more than the current computed maximum value. This type of pruning allows a great reduction of the IO cost on cube accesses.
2. While the order of the sub-cube visitation for the range-sum query is not very important in terms of IO accesses, it is no longer true in the case of range-max query. Due to the possibility of pruning some searching processes, it is highly beneficial to find a correct order of the evaluation of the sub-range queries so as to increase the probability that a sub-range can be pruned.
3. A maximum data is not just a result of an aggregation function, it is also a data that appears in the data cube. As a result, a maximum data can associate with the location of the data cube cell where the maximum appears. Using the location, some of the range-max query can be done much faster. For example, if we know that the overall maximum is at location <3,8,4>, then any range-max query that includes <3,8,4> can be answered in just one cube access --- the access of the cell <3,8,4> itself. In this paper, we shall formulate our algorithm to use this location to further decrease the access cost of range-max query.

2 Hierarchical Compact Cube

Definition 2.2 A data cube *DC* of d dimension, is a d-dimensional array. For each dimension, the index can be ranged from 0 till s_i-1 inclusively. We will denote s_i as the size of the i^{th} dimension. In this paper, a cell in the data cube can be expressed in the following form,

$$DC\langle x_1,\ldots,x_n\rangle \quad \text{where } 0 \le j_i < s_i$$

Example 2.1 Figure 2.1 shows a data cube of 2 dimension. The size of the first dimension (represented as row in this paper) is 5, and the size of the second dimension (represented as column) is 7.

	0	1	2	3	4	5	6
0	**5**	**24**	**17**	**32**	**9**	**21**	**34**
1	**30**	**11**	**2**	**20**	**25**	**8**	**14**
2	**16**	**26**	**1**	**13**	**15**	**3**	**28**
3	**31**	**4**	**29**	**6**	**33**	**18**	**28**
4	**23**	**22**	**12**	**19**	**10**	**27**	**35**

Figure 2.1 A data cube

Definition 2.2 Given a data cube DC of d dimension, and $\underline{d}$ integers m_1 ,…, m_d, the **compact cube** of DC, denoted as CC, is another data cube such that

1. it has the same dimension d, and
2. if the size of the i^{th} dimension in DC is $\underline{s_i}$, i.e., it ranges from 0 to s_i-1, then the dimension i in CC will be ranged from 0 to $\left\lfloor \frac{s_i - 1}{m_i} \right\rfloor$.
3. Each cell $CC\langle x_1,\ldots,x_d\rangle$ in CC stores two items
   - The maximum of all the cells $DC\langle j_1,\ldots, j_d\rangle$ where $m\, x_i \le j_i < \min(m_i(x_i+1), s_i)$ and
   - the position of one of the cells that holds this maximum value.

In this paper, we shall denote the maximum value stored in $CC\langle x_1,\ldots,x_d\rangle$ simply as $CC\langle x_1,\ldots,x_d\rangle$.value, and the maximum location as $CC\langle x_1,\ldots,x_d\rangle$.location. For simplicity's sake, we assume $m_1 = \ldots = m_d = m$, and we shall call this integer m the **compact factor** of the compact cube. However, our algorithm is equally applicable when m_i are not the same.

Example 2.2 Figure 2.2 shows a compact cube of the data cube shown in Figure 2.1. The compact factor is set to 2. Note that among the data in $DC\langle i, j\rangle$ where $0 \le i, j \le 1$, the maximum value is 30, and it appears in location <1,0>. This information is stored in CC<0,0> of the compact cube. Note that for CC<1,3>, the maximum value 28, can be derived from DC<2,6> and DC<3,6>. Our compact cube just randomly picks one of these locations and stores it. Note also that CC<2,3> only summarises the maximum of only one cell in the data cube: DC<4,6> and thus CC<2,3>.value is exactly equal to DC<4,6>.

	0	1	2	3
0	**30** *1 0*	**32** *0 3*	**25** *1 4*	**34** *0 6*
1	**31** *3 0*	**29** *3 2*	**33** *3 4*	**28** *2 6*
2	**23** *4 0*	**19** *4 3*	**27** *4 5*	**35** *4 6*

Figure 2.2 A Compact Cube

Definition 2.3 Given a compact cube CC of dimension d, and d integers m_1 ,…, m_d, its **compact cube**, CC_2, is another compact cube such that

1. it has the same dimension d, and
2. if the dimension i in CC is ranged from 0 to s_i-1, then the dimension i in CC_2 will be ranged from 0 to $\left\lfloor \frac{s_i - 1}{m_i} \right\rfloor$.
3. Each cell $CC_2\langle x_1,\ldots,x_d\rangle$ in CC_2 stores two items
   - The maximum of $CC\langle j_1,\ldots, j_d\rangle$.value where $m\, x_i \le j_i < \min(m(x_i+1), s_i)$ and
   - the location attribute of one of the cells which holds this maximum value.

To simplify the discussion in this paper, we shall again assume that all m_i are the same, and likewise refer it as the **compact factor**.

Example 2.4 With the compact cube CC as shown in Figure 2.2, we can compact to generate another compact cube CC_2. With compact factor to be 2, CC_2<0,0> contains the maximum value among CC<0,0>, CC<0,1>, CC<1,0> and CC<1,1>. From Figure 2.2, we can conclude that the maximum value is 32, and it is at CC<0,1>. Hence, CC_2<0,0>.value will be 32. CC_2<0,0>.location will be equal to the location attribute of CC<0,1>, i.e., <0,3>. The completed CC_2 is shown in Figure 2.3.

	0	1
0	**32** *0 3*	**34** *0 6*
1	**23** *4 0*	**35** *4 6*

Figure 2.3
A rank 2 compact cube

Definition 2.4 Given a data cube DC and an integer m, an **Hierarchical Compact Cube** denoted by HC, is a sequence of compact cubes CC_0,…, CC_h such that

1. CC_0 is the data cube DC itself.
2. CC_k ($k \ge 1$) is the compact cube of CC_{k-1} with compact factor m.
3. CC_h is the only compact cube which contains only one single cell.

We shall call the integer m the **compact factor** of the hierarchical compact cube HC, h the **height** of the HC. We shall refer CC_i as the rank i compact cube of HC and CC_h also as the **topmost** compact cube of HC.

Example 2.4 With the data cube as shown in Figure 2.1, we can construct a hierarchical compact cube HC. The rank 0 compact cube is the data cube itself. The rank 1 compact cube is shown in Figure 2.2, and the rank 2 compact cube is shown in Figure 2.3. Lastly, Figure 2.4 shows the rank 3, the topmost compact cube, which results from compacting the rank 2 compact cube.

	0
0	**35** *4 6*

Figure 2.4
A rank 3 compact cube

Definition 2.5 A max-range query with respect to a data cube *DC* of dimension *d* can be specified as

$$[< L_1 ,\ldots, L_d > , < H_1 ,\ldots, H_d >]$$

such that for each dimension *i*, $0 \leq L_i < H_i \leq s_i$ where s_i is the size of the i^{th} dimension of *DC*. The query returns the maximum value among all the data in $< x_1 ,\ldots, x_d >$ with $L_i \leq x_i < H_i$.

Example 2.5 In Figure 2.5, the shadowed area represents the range [<1,1>, <4,5>].

	0	1	2	3	4	5	6
0	5	24	17	32	9	21	34
1	30	11	2	20	25	8	14
2	16	26	1	13	15	3	28
3	31	4	29	6	33	18	28
4	23	22	12	19	10	27	35

Figure 2.5 The range [<1,1>, <4,5>]

Definition 2.6 Given a cell $CC_r<x_1 ,\ldots, x_d>$ of a r^{th} rank compact cube with compacting factor *m*, a region $R = [<L_1 ,\ldots, L_d > , <H_1 ,\ldots, H_d >]$ is said to be contained in the cell if and only if for each *i* ($1 \leq i \leq d$),

1. $m^r x_i \leq L_i$ and
2. $H_i \leq \min(m^r(x_i+1), s_i)$.

where s_i is the size of the i^{th} dimension of the compact cube CC_r. The region *R* is said to be a **full region** with respect to the cell $CC_r<x_1 ,\ldots, x_d>$ if all the equality signs in both conditions hold. Otherwise, *R* is called a **partial region** with respect to the cell $CC_r<x_1 ,\ldots, x_d>$.

Example 2.6 Refer to the hierarchical compact cube *HC* as described in Example 2.4. The region [<0,0>,<4,4>] is contained in CC_2[0,0] as $2^2*0 \leq 0$ and $4 \leq 2^2*1$. Indeed, as both the equality signs hold, the region is also a full region. The same region is also contained in CC_3<0,0> as $2^3*0 \leq 0$ and $4 \leq 2^3*1$. However, the region is only a partial region with respect to CC_3<0,0> as the second equality does not hold. Finally, the region is not contained in CC_1<0,0> as the second condition "$4 \leq 2^1*1$" fails.

3 Using the Hierarchical Compact Cube for range query

Before we present the algorithm to handle range-max query, we shall illustrate the idea behind using the following example:

Example 3.1 Refer to the data cube as described in Example 2.1, we want to find the maximum value in the range *R* = [<1,1>, <5,5>]. This range is shown in the shadow area of Figure 3.1.

	0	1	2	3	4	5	6
0	5	24	17	32	9	21	34
1	30	11	2	20	25	8	14
2	16	26	1	13	15	3	28
3	31	4	29	6	33	18	28
4	23	22	12	19	10	27	35

Figure 3.1 A sample query

Instead of accessing the data cube directly to find the maximum, we will first look at the topmost rank of the hierarchical compact cube, the rank 3 compact cube. This compact cube is shown in Figure 3.2. The dotted rectangle represents the region R (ranged [<1,1>, <5,5>]) wrt the Rank 3 compact cube (ranged [<0,0>, <5,7>]).

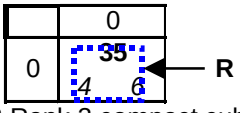


Figure 3.2 Rank 3 compact cube

This compact cube cell reveals that the maximum within the region [<0,0>, <5,7>] is 35 and it is in the location <4,6>. Given any region R that is contained in CC_3<0,0>, there are three possibilities,

1. *R* is a full region with respect to CC_3<0,0>,
2. *R* is a partial region, but the maximum cell *DC*<4,6> is inside *R*,
3. *R* is a partial region, and the maximum cell *DC*<4,6> is not inside *R*.

In either case 1 or case 2, as the maximum element in the cell *DC*<4,6> is also inside *R*, the region *R* contains the maximum value. We can then return 35 as the answer immediately and do not need to do any further investigation. Only in case 3 do we need to investigate further. In this example, *R* belongs to case 3.

We now apply the bound and branch [1] and the divide and conquer idea [2] to subdivide the region R into m^d sub-regions. In this example, it is divided into $R1$, $R2$, $R3$ and $R4$ so that each sub-region is contained in exactly one rank 2 compact cube cell. This is shown in Figure 3.3. The original query can now be transformed into four sub-queries to find the maximum values of region $R1$, $R2$, $R3$ and $R4$, and the final result is the largest of these four maximums.

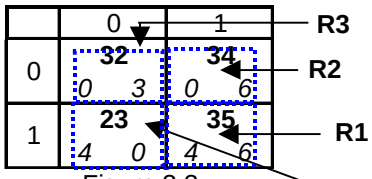


Figure 3.3
A rank 2 compact cube

While the final answer is independent of which four sub-queries is evaluated first, however, if we compute $R1$ and discovers that the maximum is indeed 35, then we can immediately prune the query on $R2$, as its maximum is at best 34. We therefore propose to compute the sub-queries in the following order:

1. All the regions that are full regions first, then
2. All the partial regions with the largest maximum evaluated first and the smallest maximum evaluated last.

The full regions can be computed without any further subdivision. Hence, they should be evaluated first. On the other hand, a partial region may need to investigate furthermore if the maximum location is not inside the partial region. Hence, they are evaluated later. In order to compute the largest maximum first, we need to maintain some sorted order of these partial regions. A complete sorting is quite expensive. For instance, in our example, if 35 is found to be the answer, it is a waste of resources to pre-sort the regions $R2$, $R3$ and $R4$. Consequently, a priority queue implemented using implicit heap is introduced to keep those "to-be-investigated" regions such that the largest cell-maximum can be immediately available in the front of the queue. Note that as we are using heap structure, we do not need all the elements in the queue completely sorted.

In this example, none of the regions $R1$, $R2$, $R3$ or $R4$ is a full region, we therefore proceed to examine the four partial regions. The first region to be investigated is region $R1$. It is contained in the compact cube CC_2<1,1> that also holds the largest possible maximum, 35. However, as 35 is at position <4,6>, it is outside the region $R1$. Hence, we cannot immediately conclude the maximum of $R1$. $R1$ is now inserted into the priority queue Q for further analysis. Similarly, regions $R2$, $R3$ and $R4$ are all partial regions and their respective maximums do not fall in their corresponding regions. Hence, they are all inserted into Q. As Q always ensures that the largest element is in the front of the queue, hence, the elements contained in Q are regions $R1$, $R2$, $R3$ and $R4$, with $R1$ being in the front of the queue.

Now we further investigate the largest element in Q, $R1$. The region can be further sub-divided into only one region $R1a$ in the rank 1 compact cube, as shown in Figure 3.4.

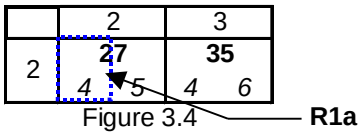


Figure 3.4
In a rank 1 compact cube

Now $R1a$ is still just a partial region, and its maximum, 27, is at position <4,5>, which is outside the region $R1a$. Hence, we again cannot conclude the maximum value of $R1a$ yet and hence $R1$. We need to insert the region $R1a$ into the queue for further processing. Now, the queue Q contains the regions $R2$(max=34), $R3$(max=32), $R1$a(max=27) and $R4$(max=23) with $R2$ being in the front of the queue.

The next region dequeued from Q is region $R2$. It can be subdivided into $R2a$ and $R2b$, as shown in Figure 3.5.

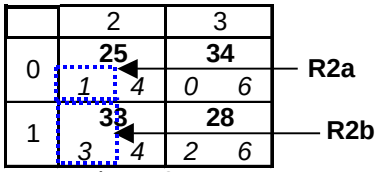


Figure 3.5
In a rank 1 compact cube

None of them is a full region. However, as the maximum value stored in CC_1<1,2>, 33, is within the region $R2b$, we can conclude that the maximum of region $R2b$ is 33. In other words, the overall maximum of the original query is *at least* 33. Now $R2a$ has a maximum value of 25, which is less than the current maximum, 33. Therefore, we can skip this region. At this moment, the current maximum is 33, and the queue Q contains regions $R3$, $R1a$ and $R4$.

The next region $R3$ has only a maximum of 32, which is smaller than the current maximum, 33. We can skip region $R3$. But since the queue Q always removes the largest element from the queue, the remaining elements in Q are even smaller and can never improve the current maximum, 33. As a result, we can stop our algorithm and conclude that the current maximum is 33.

The following summarizes our algorithm:

Algorithm 3.1 [Maximum Query]

Let DC be a given data cube and let q be the smallest domain value of the measure attribute. Let HC be the hierarchical compact cube of DC with compacting factor m and height h. We find the result of a range-max query R_0 by the following steps:

1. Let Q be an empty priority queue, which stores tuples of the form $[R, max_{guess}, h_t]$ where R is a range, max_{guess} is an estimated maximum of the range R, and h_t is the smallest height of all the compact cubes in HC that range R has investigated.
2. To start with, if R_0 covers the entire the data cube, then the topmost compact cube, CC_h, is exactly R_0. We return the maximum value stored in CC_h<0,…,0> as the query result and exit the algorithm.
3. Otherwise, we insert $[R_0, q, h]$ inside the priority queue Q. The queue is inserted in a way that a larger max_{guess} will be dequeued first, and the smaller max_{guess} will be dequeued later. We also initialise the current maximum max_{cur} as $q - 1$. We now perform the following processes:
4. If Q is empty, then stop the algorithm, and report max_{cur} as the actual maximum.
5. Otherwise, dequeue the largest item $[R, max_{guess}, h_t]$ from the priority queue Q. If the max_{guess} is not more than max_{cur}, stop the algorithm, and report max_{cur} as the actual maximum.
6. Let $\{C_j\}$ be the minimum set of rank $(h_t - 1)^{th}$ compact cubes such that $\bigcup C_j$ covers R. For each j, we denote R_j as the subregion of R that C_j overlaps. In other words, $R_j = R \bigcap C_j \neq f$.
7. For each R_j such that R_j is a full region with respect to the compact cube C_j, we query the maximum value stored in the corresponding $(h_t - 1)^{th}$ compact cube C_j, which is exactly the maximum of R_j. If the returned value is more than the current maximum max_{cur}, we update max_{cur} to be the returned value.
8. For the rest of R_j that is only a partial region with respect to the compact cube C_j, we query the $(h_t-1)^{th}$ compact cube to find the maximum of C_j, max_{query}. This value gives the upper bound of the maximum of R_j. We have three cases:
   a. If the returned value max_{query} is not more than the current maximum max_{cur}, then the actual maximum of R_j cannot be more than max_{cur} and we can skip this region. We repeat step 8 for another region $R_{j'}$.
   b. On the other hand, if the returned value is more than the current maximum and if the maximum location is inside R_j, then we confirm that the maximum value of R_j is indeed max_{query}. We update max_{cur} to be max_{query} and continue step 8 with another region $R_{j'}$.
   c. Finally, if the returned value is more than the current maximum, and the maximum location is outside R_j, we need to do further investigation on R_j to confirm its actual maximum. We insert the item $[R_j, max_{query}, h_t-1]$ into Q.
9. After all R_j have been processed, we repeat step 4 of the algorithm until Q is empty.

4 The constant-time average access cost of our method

In this section, we shall first formulate a recurring equation on the average number of compact cube accesses. We then prove that the average number of cube accesses is bounded by a constant that is independent of the size of the compact cube. To start with, we note that during the searching of the maximum value at the r^{th} rank compact cube, the total cost $cost_r$ can be divided into two parts:

1. The query of the maximum values of all the immediate children of the r^{th} rank compact cube, as required in step 7 and step 8 of the Algorithm 3.1. We can assume that there are N such $(r - 1)^{th}$ rank children.
2. The possible further query on these N children as described in step 8, part (c) of the Algorithm 3.1.

If k_r is the expected number of children that are required to perform further query, then

$$cost_r = N + k_r\, cost_{r-1}$$

To estimate N, we assume that during the query R on the r^{th} rank compact cube, the r^{th} rank compact cube covers exactly w_i $(r - 1)^{th}$ rank compact cubes in the i^{th} dimension where $1 \le w_i \le m$. Clearly, the r^{th} rank compact cube covers exactly

$$N = \prod_{i=1}^{d} w_i$$

$(r - 1)^{th}$ rank compact cubes. We denote the sub-regions that these compact cubes cover to be $R_1, \ldots, R_N$. Note that according to Algorithm 3.1, during the processing of any region R of the r^{th} rank cube, we need to access the maximum value stored inside all the sub-cubes R_j in step 7 and 8 of the algorithm. Hence, our algorithm needs to access exactly N compact cubes of rank $(r - 1)$.

Given that the compact factor is m, each w_i will be ranged between 1 and m. Thus, the value of N, in the worst case, is at most m^d, which is a constant. Note that in average, the expected value of N is much smaller. If either the starting value or the ending value for the i^{th} dimension of the given range is a random variables, then expected value of each w_i can be shown to be only about $m/2$. Thus,

the expected value of N is $1/2^d$ smaller than the worst case. In conclusion, the number of the $(r-1)^{th}$ rank compact cubes needed to be investigated from a r^{th} rank compact is bounded by the constant m^d in the worst case.

To estimate exactly the value of k_r is much more complex. However, we can show that the value of k_r approaches to 0 for lower rank compact cube as the data cube size increases. According to the step 8 of our algorithm, it is required that a sub-region R_p will be inserted into the queue Q for further investigation only if the following conditions are satisfied:

1. The children R_p is a partial region, and
2. the returned maximum on query of R_p is more than the current maximum, and
3. the location of the maximum is not inside R_p.

There can be plenty of compact cubes can do not satisfy the first condition. As illustrated in Figure 4.1, given a region R in the d-dimension r^{th} compact cube, in the worst case, there are only at most $\prod_{i=1}^{d} w_i - \prod_{i=1}^{d} (w_i - 2)$ partial regions in the $(r-1)^{th}$ compact cube where w_i is the length of the i^{th} dimension that R overlaps with the $(r-1)^{th}$ compact cube. The proportion of partial regions is even smaller when the compact factor m increases, as well as when some dimension ranges falls exactly at the division mark (as shown on the row 5 in figure 4.1) which frequently occurs in lower rank compact cubes.

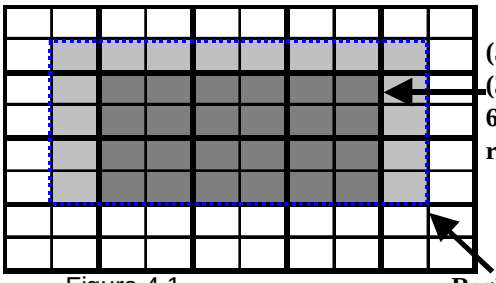


Figure 4.1
Illustration on the numbers of full sub-regions.

To satisfy the second condition, we note that Algorithm 3.1 will first compute the maximum of all the full regions first in step 7. The probability that a partial block R_p has a maximum more than the current maximum is the probability that among all the "explored" regions and the partial block, the largest value is at that partial block. Now, when we start our algorithm by first investigating the topmost rank h compact cube, there are at least $\prod_{i=1}^{d} (w_i - 2)$ sub-regions covered by some rank $(h\text{-}1)^{th}$ compact cube being investigated. Each such rank $(h\text{-}1)^{th}$ compact cube contains about $m^{(h-1)d}$ data. As a result, we can assume that at least $\left(\prod_{i=1}^{d} (w_i - 2)\right) m^{(h-1)d}$ data has been explored during the visit of these full regions covered by these rank $(h\text{-}1)^{th}$ compact cubes. Subsequently, for the remaining partial regions, some full regions of lower rank cubes will also be explored. This further increases the "explored" area and thus decreases the chance that the maximum is found in R_p. However, for simplicity sake, we shall ignore these surpluses in this analysis. In other words, we only assume that at least

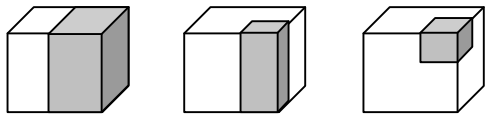
$$\left(\prod_{i=1}^{d} (w_i - 2)\right) m^{(h-1)d}$$

data has been explored before accessing the partial block R_p. The size of the partial block of r^{th} rank compact cube is about m^{rd}. Consequently, the probability that the first R_p contains a larger maximum than the current maximum is not more than

$$\frac{m^{rd}}{\left(\prod_{i=1}^{d} (w_i - 2)\right) m^{(h-1)d}}$$

Finally, even if the second condition is satisfied --- R_p has a cell maximum that is greater than the current maximum, as long as this maximum is in the region R_p, it does not fulfil the third condition. In this case, we need not do any further investigation. To estimate this probability, we first illustrate the computation using d=3 case. For a partial region in a compact cube of size m, it can fall into three different cases:

1. The region is on the surface. There are $\binom{3}{1} 2^1 (m-2)^2$ such regions. The probability that a chosen point is in the region is ½.
2. The region is on the edge of the cube. There are $\binom{3}{2} 2^2 (m-2)^1$ such regions. The probability that a chosen point is in the region is ¼.
3. Finally, the region can be on the corner of the cube. There are $\binom{3}{3} 2^3 (m-2)^0$ such regions. The probability that a chosen point is in the region is 1/8.



We can generalise the sum for any dimension d, the probability that a particular point is in a partial region is

$$\frac{\sum_{k=1}^{d}\binom{d}{k}(m-2)^{d-k}}{\sum_{k=1}^{d}\binom{d}{k}2^{k}(m-2)^{d-k}} \approx \left(1-\frac{1}{m}\right)^{d}$$

Hence, we can deduce that the expected value of k_r is not more than

$$\left(\prod_{i=1}^{d} w_i - \prod_{i=1}^{d}(w_i-2)\right)\left(\frac{m^{rd}}{\left(\prod_{i=1}^{d}(w_i-2)\right)m^{(h-1)d}}\right)\left(1-\frac{1}{m}\right)^{d}$$

Since the value of m, w_i, and d are all independent on the size of the original data cube, we can simply rewrite the above expression as,

$$cm^{(r+1-h)d}$$

where the expected value of the constant c only depends on the value of d and m. With the bound of k_r, we have

$$\text{cost}_r < m^d + cm^{(r+1-h)d}\,\text{cost}_{r-1}$$

Expanding the sum, and putting $r = h$, we have,

$$\text{cost}_h < m^d\left(1+cm^d+c^2m^d+c^3+\frac{c^4}{m^{2d}}+\frac{c^5}{m^{5d}}+\cdots\right)$$

Note that the sum at the right hand infinite sum converges to a fix number. In other words, for any arbitrary large data cube, the total number of cell accesses, $cost_h$ is bounded by a constant, which is independent on the size of the data cube.

4.1 Experiment Result

The following figures show some of our experiment results.

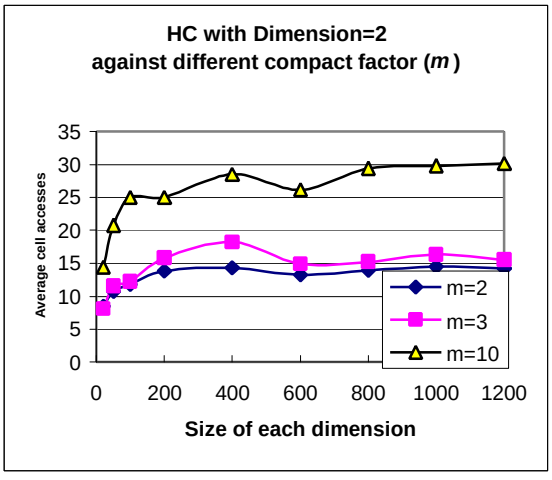


Figure 4.1 Impact of cube size for different compact factor for 2-D Cube

We generated a set of hierarchical compact cubes by varying the data size, compact factor and dimension independently. For simplicity, we consider data cubes with equal sized dimension. We then generate about 100,000 queries of random size and measure the average cell accesses required. The experiment is run in Linux Red hat 6.0 and several observations can be concluded:

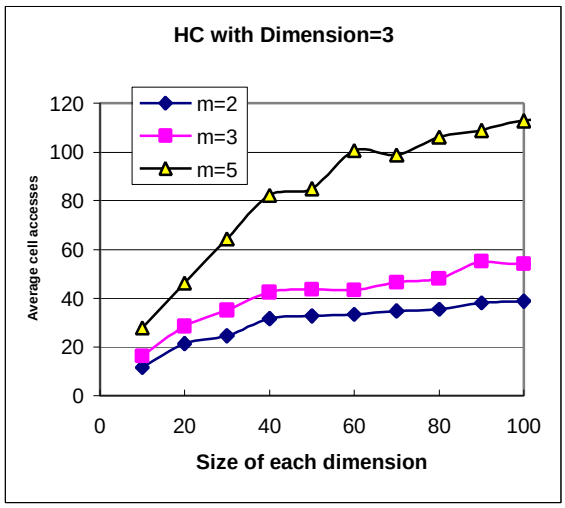


Figure 4.2 Impact of cube size for different compact factor for 3-D Cube

1. From Figure 4,1 and Figure 4.2, there are strong evidences that the average number of cell accesses does converge to a constant when the size of the data cube increases. For large set of data, the performance is not dependent on the number of data in the data cube. This coincides with our analysis. Furthermore, the convergent rate is faster for smaller compact factor and lower dimension.
2. The performance also improves when the compact factor decreases. The best compact factor, as shown in both Figure 4.1 and 4.2, is 2.

As shown in Figure 4.3, the average number of cell accesses grows exponentially as the dimension increases. This also coincides with the factor m^d shown in the analysis result.

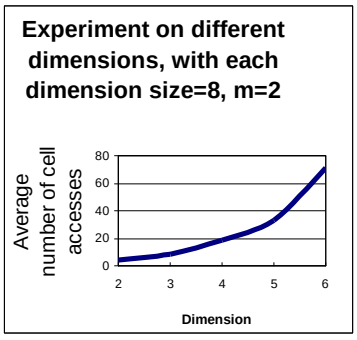


Figure 4.3
Impact of dimension on the overall performance

5 Updates and storage costs of the hierarchical compact cube

When we update a data in the data cube, we may need to update also the hierarchical compact cube. As mentioned in [6,13], our hierarchical compact cube is imperfect if it incurs a huge update cost. Likewise, our method should not incur too much extra storage costs. In this section, we shall show that the maintenance cost of the hierarchical compact cube containing N data is only $O(\log_m N)$. Furthermore, the extra storage cost is a factor smaller as compared to range-sum query methods. [9,13]

There are two types of update to the data cube. We can either increase a value or to decrease a value of a data cube cell. These two updates require a different average update cost analysis on the hierarchical compact cube.

5.1 Maintenance cost for increment

In the case of increment of a cell c in a r^{th} rank compact cube, if the increased value does not exceed the overall maximum of the $(r+1)^{th}$ rank compact cube that c belongs to, then no further update is required. The total update cost is to access the cell c itself, and to query the overall maximum by accessing one cell of the $(r+1)^{th}$ rank compact cube. On the other hand, if the increment affects the overall maximum (for instance, the update is to increase the actual largest value), then the cell of the $(r+1)^{th}$ rank compact cube which contains the overall maximum needed to be updated also. This propagates the update to the $(r+1)^{th}$ rank compact cube, and we now need to query the $(r+2)^{th}$ rank compact cube recursively. The propagation will stop when the update does not affect the maximum stored in its parents or in the worst case, r is the height of the hierarchical compact cube. In other words, in the worst case, the update cost is h. Given that m is the compact factor of the hierarchical compact cube, and d is its dimension, the total number of data in the data cube N, is about m^{hd}. Hence, the update cost h is about

$$\frac{1}{d}\log_m N$$

In other words, for a fixed dimension, the worst case increment cost is only $O(\log_m N)$. The average update case, however, is only a constant. An update of cell c is propagated only when the updated value overtakes the overall maximum. Given that the cell being increased is the k^{th} largest cell, we can assume that with only probability $1/k$, the value of this k^{th} largest cell is increased to overtake the maximum. By summing k from 1 to m^d, and assuming that each cell is updated with the same probability $1/m^d$, the probability that a cell is increased to overtake the overall maximum, and thus propagation to its parent is required, is about

$$\frac{d\ln m + g}{m^d}$$

where γ is the Euler's constant (=0.5771..).

As this probability is independent of the rank, hence, an update of a cell in the data cube (a rank 0 cell) can be propagated to a rank 1 parents cell has the same probability that this update will be further propagated to rank 2. It is a geometric progress, and the expected number of cells that requires update is

$$1\times 1 + 2\times\frac{d\ln m + g}{m^d} + \cdots + h\left(\frac{d\ln m + g}{m^d}\right)^{h-1} < \frac{m^{2d}}{\left(m^d - d\ln m - g\right)^2} = 1 + O\left(\frac{d\ln m}{m^{2d}}\right)$$

In other words, the average increment cost is bounded by a constant, regardless of the size of the data cube. Furthermore, for data cube with high dimension, the average number of cells that needed to be updated triggered by an increment operation is very closed to 1.

5.2 Maintenance cost for decrement

In the case of decrement of a data cube cell, there are two different cases. If the decrement cell does not appear as a maximum value in some of the compact cubes, then no update on the hierarchical compact cube is required. The total update cost is to access the cell c itself, and to verify that it indeed does not appear as maximum in any compact cube query by an one-cell access of its parent rank 1 compact cube. On the other hand, if the decrement cell appears to be a maximum of its parent compact cube, then the cell c itself may not necessarily remain to be the overall maximum. We need to access all the siblings cells of c in the rank 0 compact cube to elect the new overall maximum. This requires an additional m^d queries. As the value of the overall maximum is changed, the update always needs to propagate to the higher level compact cube. The propagation is done recursively until the updated cell is not the maximum cell held by its parents, or in the worst case, when we reach the topmost compact cube. This gives us the average cost to be h cell accesses and the worst cost to be $h\ m^d$ cell accesses. Both the average and worst case decrement costs are $O(\log_m N)$.

5.3 Extra Storage cost

Finally, although our method needs to store a set of compact cubes of different levels, the overall storage cost is still acceptable. For an hierarchical compact cube such that d dimensions are compacted with compacting factor m, the overall number of extra compact cube cells is only $\frac{1}{m^d - 1}$ of the number of data in the data cube. As compared to the prefix sum method where the prefix sum cube is as big as the underlying data cube, our method has a far small extra storage cost then many existing methods [9,12,13,14].

6 Conclusion

Due to an increasing demand for OLAP and data cube applications, efficient calculation of range queries such as the range-max queries has become more important in recent years. Several pre-computations and indexing techniques have been developed to answer the range-sum queries efficiently, but these methods may not be able to apply to the case of range-max. In this paper, we propose the hierarchical compact cube method for processing the range-max queries. We have explored and incorporated the following ideas into our method:

1. We employ an *hierarchical structure* that, applying bound-and-branch as well as divide-and-conquer techniques in multidimensional data, allows an efficient incremental refinement to query the maximum value of any arbitrary range-max query.
2. Different from the range-sum query, we observe that order of the sub-ranges investigation has a huge impact on the overall performance of the query. We propose to use a *priority queue* implemented using heap to store unprocessed regions. This partial ordering process is proven to greatly improve the performance of our algorithm.
3. We introduce the *maximum-location attribute* to further improve the performance of a range-max query. This location allows many early pruning of unnecessary searches.

Both the analysis and experiment results show that our method provides in average a constant time evaluation of range-max queries, and yet incurs only a low $O(\log_m N)$ update cost. Finally, the extra storage requirement for the hierarchical compact cube is also much smaller as compared to the prefix cube used in many efficient range-sum queries algorithms.

References

[1] L. Mitten. "Branch and bound methods: General formulation and properties" in Operations Research, 18:24-34, 1970.

[2] Jon Louis Bentley. "Multidimensional divide and conquer". in Comm. ACM, 23(4):214-229,1980

[3] E.F. Codd, "Providing OLAP (on-line analytical processing) to user-analysts: an IT mandate". Technical report, E.F. Codd and Associates, 1993.

[4] Ashish Gupta, Venky Harinarayan, Dallan Quass, "Aggregate-query processing in data warehousing environments". In Proceedings of the 18th Inernational Conference on Very Large Databases, pages 358-369, Zurich, Switzerland, September 1995.

[5] Venky Harinarayan, Anand Rajaraman. Jeffrey D. Ullman. "Implementing Data Cubes Effieciently". In Proceedings of ACM SIGMOD 1996 International Conference on Management of Data, Montreal, Canda, June 1996 pages205-216.

[6] Jim Gray, Adam Bosworth, Andrew Layman, Hamid Pirahesh. "Data cube: A relational aggregation operator generating group-by, cross-tabs and sub-totals". In Proceedings of the 12th International Conference on Data Engineering, pages 152-159, 1996

[7] The OLAP Concil. "MD-API the OLAP Application Program Interface Version 0.5 Specification", September 1996.

[8] Sameet Agarwal, Rakesh Agrawal, Prasad M. Deshpande and Ashish Gupta, "On the Computation of Multidimensional Aggregates", In Proceedings of the 22nd VLDB Conference, Bombay, India, September 1996, pages 506-521.

[9] Inderpal Singh Mumick, Dallan Quass, Barinderpal Singh Mumick, "Maintenance of Data Cubes and Summary Tables in a Warehouse", In Proceedings of ACM SIGMOD 1996 International Conference on Management of Data, June 1997, pages 100-111.

[10] Rakesh Agrawal, Ashish Gupta. Sunita Sarawagi. "Modeling multidimensional databases". In Proc. of the 13th International Conference on Data Engineering, Birmingham, U.K., April 1997.

[11] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, Ramakrishnan Srikant. "Range Queries in OLAP Data Cubes". In Proceedings of the ACM SIGMOD Conference on the Management of Data, pages 73-88,1997.

[12] Steven Geffner, Divyakant Agrawal, Amr El Abbadi, Terence R. Smith. "Relative Prefix Sum: An Efficient Approach for Querying Dynamic OLAP Data Cubes". In Proceedings of the 15th International Conference on Data Engineering, pages 328-335, 1999

[13] Chee Yong Chan, Yannis E. Ioannidis. "Hierarchical Cubes for Range-Sum Queries". Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999 pages 675-686.

[14] Hua-gang Li, Tok Wang Ling, Sin Yeung Lee, "Range-Max/Min Query in OLAP Data Cube". Appear in Proceedings of the 11th DEXA Conference, Greenwich, 2000.